INL/EXT-13-29568

RELAP5-3D Restart and Backup Verification Testing

George L Mesina

September 2013

The INL is a U.S. Department of Energy National Laboratory
operated by Battelle Energy Alliance

INL/EXT-13-29568

RELAP5-3D Restart and Backup Verification Testing

George L Mesina

September 201

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Naval Reactors
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

RELAP5-3D Restart and Backup Verification Testing

INL/EXT-13-29568

September 2013

George Mesina — 9/23/13

Dr. George Mesina
Modeling and Simulation Engineer
Nuclear System Design and Analysis Division

Reviewed by

Cliff Davis — 9/23/13

Cliff Davis
Nuclear Engineer
Nuclear System Design and Analysis Division

James R. Wolf for — 9/23/13

J. Hope Forsmann
Computer Scientist
Software Services Division

Nolan Anderson — 9-23-13

Nolan Anderson
Nuclear Engineer
Nuclear System Design and Analysis Division

James R. Wolf — 9/23/13

Dr. James Wolf
Project Manager
Nuclear System Design and Analysis Division

EXECUTIVE SUMMARY

Existing testing methodology for RELAP5-3D employs a set of test cases collected over two decades to test a variety of code features and run on a Linux or Windows platform. However, this set has numerous deficiencies in terms of code coverage, detail of comparison, running time, and testing fidelity of RELAP5-3D restart and backup capabilities.

The test suite covers less than three quarters of the lines of code in the relap directory and just over half those in the environmental library. Even in terms of code features, many are not covered. Moreover, the test set runs many problems long past the point necessary to test the relevant features. It requires standard problems to run to completion. This is unnecessary for features can be tested in a short-running problem. For example, many trips and controls can be tested in the first few timesteps, as can a number of fluid flow options.

The testing system is also inaccurate. For the past decade, the diffem script has been the primary tool for checking that printouts from two different RELAP5-3D executables agree. This tool compares two output files to verify that all characters are the same except for those relating to date, time and a few other excluded items. The variable values printed on the output file are accurate to no more than eight decimal places. Therefore, calculations with errors in decimal places beyond those printed remain undetected.

Finally, fidelity of restart is not tested except in the PVM sub-suite and backup is not specifically tested at all. When a restart is made from any midway point of the base-case transient, the restart must produce the same values. When a backup condition occurs, the code repeats advancements with the same timestep. A perfect backup can be tested by forcing RELAP5 to perform a backup by falsely setting a backup condition flag at a user-specified-time. Comparison of the calculations of that run and those produced by the same input w/o the spurious condition should be identical. Backup testing is more difficult than the other kinds of testing described above and it requires additional coding to implement.

The testing system constructed and described in this document resolves all of these issues. A matrix of test features and short-running cases that exercise them is presented. A small information file that contains sufficient data to verify calculations to the last decimal place and bit is produced. This testing system is used to test base cases (called null testing) as well as restart and backup cases. The programming that implements these new capabilities is presented.

CONTENTS

TABLES

FIGURES

ACRONYMS

BC	Boundary Conditions
CCFL	Counter Current Flow Limiting
CHF	Critical Heat Flux
CPU	Central Processor Unit
DA	Developmental Assessment
ECC	Emergency Core Coolant
HSE	Hydro Static Equilibrium
HTC	Heat Transfer Coefficient
INL	Idaho National Laboratory
INEEL	Idaho National Engineering and Environmental Laboratory
KB	Kilobyte
LOCA	Loss Of Coolant Accidents
MB	Megabyte
PVM	Parallel Virtual Machine
RELAP	Reactor Excursion and Leak Analysis Package
RHS	Right Hand Side
SDIVD	Software Development, Implementation and Verification Document
TH	Thermal Hydraulics
V&V	Verification and Validation

SYMBOLS

English

b	RHS of Δp Linear System
di	Probability of detecting differences
H_0	Null Hypothesis
H_1	Alternative Hypothesis
N	Number of test cases
P	Probability function
S	Size of the verification file
t_i	Number of timesteps if the i^{th} input case
T	Temperature
T_r	Trip
UP	User-reported Problem
u_f	Liquid internal energy
u_g	Gas internal energy
V_f	Velocity of the Liquid (Fluid)
V_g	Velocity of the Gas
X	Random Variable for the Hypothesis Test that two runs are identical
X_a	Noncondensable Quality
X_i	Random Variable for Test Case i, the i^{th} input deck.
Y	Control Variable
3D	Three dimensional

Greek

α	significance level of a hypothesis test
α_g	Void fraction of gas
β	Probability of committing Type II error in a hypothesis test
Δp	Pressure Drop
Δt	Timestep for Hydrodynamic Advancement
Δt_{kin}	Timestep for Neutron Kinetics Advancement
ε	Sum of RELAP5-3d estimate errors
φ	Neutron Flux
$\boldsymbol{\rho}_b$	Density of Boron

1. INTRODUCTION

The state-of-the-art nuclear reactor system safety analysis computer program developed at the Idaho National Laboratory (INL), RELAP5-3D[1-1], continues to adapt to changes in computer hardware and software and to develop to meet the ever-expanding needs of the nuclear industry. To continue at the forefront, code testing must evolve with both code and industry developments.

A Developmental Assessment[1-2], a form of validation testing, was performed in 1990 and a draft report written. Thereafter however, no element of Verification and Validation testing was performed for many years. In the late 1990s, verification testing was checking that all problems in the test suite ran without error messages to completion on UNIX computer platforms of five different vendors. In the early 2000s, the diffem script[1-3] was developed to check that output generated by two different code versions were character-for-character identical, except for date, time-stamp, CPU run time and anything else unrelated to actual calculations.

However, the diffem script was not the final authority. A comparison between output files was also deemed acceptable if the differences diffem found were justified by expected changes due to bug fixes or code development. If all output files compared acceptably between two versions, the new version was called acceptable. If a code version passed the limited internal test set comparison, it was released internally. Additional tests were run before the code was released externally.

The diffem approach has limitations. Typical 64-bit floating point variables in the code have values good to about 14 decimal places. In the printed output file, only to 5-8 decimal places are represented. Differences in RELAP5-3D calculations beyond eight decimal places cannot be detected by comparing two such files. Some code bugs, therefore, went undetected in RELAP5-3D product testing. Bugs were found by users through running transients of sufficient duration to expose accumulated differences in the printed output file.

In 2010, the "-stat" command line option was added to create a file of statistics to record other useful data about a run. Comparison of two stat files provides additional verification, but it is insufficient to guarantee all differences in the calculations between two code versions are discovered. In order to do that, information that is accurate to the last bit of the calculated floating point numbers must be recorded. Only then can comparisons that are 100% certain be made.

Another inadequacy in the installation test procedure is the test cases that comprise it. The RELAP5-3D installation test suite is a collection of problems that run automatically when the code is built on a computer platform. Although the test suite has increased multifold in the past two decades, coverage analysis shows that over 25% of the lines of code in the RELAP5-3D source code directory and over 40% of those in the environmental library are not exercised by the test suite. In fact, not all of the important code features are tested.

Most test cases in the test suite are run through transients far longer than necessary for testing the specific features being checked. Most features can be tested in short-running test cases. Examples include most trips and controls. A larger collection of shorter-running problems would be able to test more important features in less time.

Finally, fidelity of code restarts and backups is not tested at all. Restart cases are run, however with the exception of the PVM installation problems, test cases do not check that a given problem run completion and its restart from an intermediate restart record produce the same calculations.

A backup is quite different from a restart. In a backup, the code repeats advancements with the same or smaller timestep. There are two reasons for this. The first is that the code detects that truncation errors on a given advancement are too large. The second occurs when, partway through an advancement, the code recognizes a condition under which a slight modification of the system of equations would have

produced a better approximation of reality. In the former situation, a repeat with a smaller timestep is needed, while in the latter, the repeat uses the same size timestep.

Thus there are three categories of testing listed in Table 1.0.1, null testing, restart testing, and backup testing.

No.	Category	Description
1	Null testing	Check that two code versions produce the same calculations
2	Restart testing	Check that a restarted run produces the same calculations as the original run
3	Backup testing	Check that the code still produces the same calculations with a forced backup

Table 1.0.1 Categories of Testing

Recently a new testing procedure has been developed at another National Laboratory[1-4]. This procedure addresses all three of these areas. Development of similar testing procedures for RELAP5-3D will produce a much more robust and reliable code.

As with all testing, the simple result it produces must be interpreted. If differences are reported, they must be examined. If they result from code maintenance, development of a new feature, or the fixing of a previously reported error, the *differences in calculations may or may not mean an error* has been discovered.

The remainder of Section 1 covers the theory of testing in these three categories. Section 2 details the verification file that holds data sufficient to spot differences in even the last floating point bit of the calculations and explains why the detection of differences is nearly complete. Section 3 covers the means of making the test powerful (reducing β) by selecting code features and test cases to provide practical coverage of code calculations. Section 4 explains restart testing. Section 5 introduces and explains backup testing. Section 6 is coding and scripting for the test. Section 7 is a Bibliography.

1.1 Statistical Statement of the Testing System

To address all the testing deficiencies listed in Section 1.0, verification testing is developed using the statistical theory of hypothesis testing. It is based on a collection of individual tests where two runs are made and the calculations of the runs are compared.

The null hypothesis of the test, H_0, is: "The two runs produce exactly the same calculations." The alternate hypothesis is that calculations are different. The statistic used to test the hypothesis for the i^{th} test case, X_i, has a value of 0 if no differences are found between the two runs. It is 1 otherwise, no matter how many differences occur. When there are N test cases, then X_i is defined for each test case, i, and X is the maximum. Applying standard statistical methods[1-5], this can be expressed mathematically as:

$$X_i = \begin{cases} 0 & if \quad exactly\ 0\ differences\ are\ found \\ 1 & if \quad at\ least\ one\ difference\ is\ found \end{cases} \tag{1.1.1}$$

$$X = \max \{X_i \mid i = 1, 2, \ldots, N\} \tag{1.1.2}$$

H_0: The two runs produce exactly the same calculations (1.1.3)

A_0: Code calculations are different (1.1.4)

Test: Accept the null hypothesis if X = 0, but reject it when X = 1. (1.1.5)

Acceptance Region: {0} (1.1.6)

Rejection Region for X: {1} (1.1.7)

The Equation numbers at the right are assigned to Definitions such as (1.1.1), Statements such as (1.1.3), Tests such as (1.1.5), and Equations. Such numbered concepts will be referred to throughout as Definitions, Statements, Tests, and Equations.

Hypothesis testing potentially commits two kinds of errors. The first, Type I Error or false positive, is the rejection of the null hypothesis when it is true. That means finding differences when there are none. The probability of committing Type I Error is denoted α and is called the level of significance of the test.

$$\alpha = P(\text{Reject } H_0 \mid H_0 \text{ is true}) \tag{1.1.8}$$

The second kind of error, called Type II Error or a false negative, is to accept the null hypothesis when it is false. That means there are differences that go undiscovered. It has probability β.

$$\beta = P(\text{Accept } H_0 \mid H_0 \text{ is false}) \tag{1.1.9}$$

The power of the test is the probability of correctly rejecting the null hypothesis when it is false. The more powerful the test, the better it is.

$$\text{Power} = 1 - P(\text{Accept } H_0 \mid H_0 \text{ is false}) = 1 - \beta \tag{1.1.10}$$

These are all standard definitions[1-5]. A summary of the hypothesis testing is diagramed in Table 1.1.1.

Table 1.1.1. Hypothesis Testing Table for Test (1.1.5)

	H_0 is true No differences exist	**A_0 is true** Differences exist
Accept H_0	*Correct* Report: "No differences"	*Type II Error* Don't find extant differences
Reject H_0	*Type I Error* Detect non-existent differences	*Correct* Report: "Differences found"

Section 1.2 presents theoretical results for Type I error. Section 1.3 discusses Type II error.

1.2 The α-Significance of the Test

For null or Category 1 testing, the test is used to detect difference between two code versions, an older one and a newer one. Typically, the newer one is a modification of the older one. The calculations performed by both codes on the same input are compared. This is done for all N input files that form the verification test suite. For the i^{th} input file, X_i is evaluated according to Equation (1.1.1). After all N values of Xi are obtained, X is calculated according to Equation (1.1.2) and a conclusion is drawn based on Test (1.1.5). *Note that this test will vary in power*, Equation (1.1.10), based on two things: how the differences are checked, and what input files go into the verification set. These issues are discussed in Sections 1.3, 2 and 3. In the current Section, test power is not considered.

The surprising result is that Test (1.1.5), theoretically at least, cannot reject a null hypothesis when it is true. The practical meaning of this is that the test, *properly implemented*, will never send code developers on a fruitless search for bugs that are not there.

THEOREM 1.2.1: For Test (1.1.5), $P(X=0 \mid H_0 \text{ is true}) = 1$.

(T*his test will <u>always</u> accept the null hypothesis when it is true.*)

PROOF: Suppose H_0 is true. Then the two code versions produce identical calculations.

Therefore, there can be no differences to detect between the calculations of the two code versions, no matter what input model is run. In particular, *0 differences are found* for any of the N input models *in the verification test set*. Thus, by definition of X_i and X in Equations (1.1.1) and (1.1.2),

$$X_i = 0, \quad i = 1, 2, \ldots, N \tag{1.2.1}$$

$$X = \max \{X_i \mid i = 1, 2, \ldots, N\} = 0. \tag{1.2.2}$$

So whenever the null hypothesis is true, X=0 and H_0 will be accepted with 100% probability.

$$P(X=0 \mid H_0 \text{ is true}) = 1 \tag{1.2.3}$$

Q.E.D.

This result shows that the theoretical test will accept the null hypothesis when true; however it does not preclude implementation issues. Programming errors could mistakenly indicate differences that are not there. It also does not imply anything about the power of the test. Theorem 1.2.1 applies equally to a test that compares only one value and a test the checks everything, as well as to a test that covers 1% of the coding or 99% of the coding.

COROLLARY 1.2.1: For Test (1.1.5), $P(X=1 \mid H_0 \text{ is true}) = 0$.

The test never commits Type I Error, (never detects non-existent differences). It has significance level, α = 0.

PROOF: $P(X=0 \mid H_0 \text{ is true}) = 1$ by (1.2.3). Therefore,
$P(X=1 \mid H_0 \text{ is true}) = 1 - P(X=0 \mid H_0 \text{ is true}) = 1 - 1 = 0$. Now by definition,
$\alpha = P(\text{Reject } H_0 \mid H_0 \text{ is true}) = P(X=1 \mid H_0 \text{ is true}) = 0$.

Q.E.D.

Test (1.1.5) applies not just to Category 1 or null testing but also to Categories 2 and 3, restart and backup testing. Category 1 has two code versions and one input file, while Category 2 and 3 tests have two input files and one code version. Restart testing also involves two auxiliary files, namely the restart and plot files. The second input file restarts the run from an intermediate record of the restart and plot files and ought to perform exactly the same calculations.

THEOREM 1.2.2: Restart Test (1.1.5) commits no Type I Error. It has significance level, $\alpha = 0$.

PROOF: Suppose H_0 is true. Then the two code runs produce identical calculations. This is true despite the involvement of auxiliary files.
As before, $X_i = 0$, for i = 1, 2, … , N and so $X = \max \{X_i \mid i = 1, 2, \dots , N\} = 0$, and it follows that $P(X=0 \mid H_0 \text{ is true}) = 1$. Following the same argument as above

$$\alpha = P(\text{Reject } H_0 \mid H_0 \text{ is true})$$
$$= P(X=1 \mid H_0 \text{ is true})$$
$$= 1 - P(X=0 \mid H_0 \text{ is true}) = 1 - 1 = 0.$$

Q.E.D.

Backup testing is performed by forcing the code to repeat an advancement with the same timestep by falsely setting a backup flag. No auxiliary files are involved. The statement of the hypothesis must be altered because the calculations performed are different; in the backup run for some advancement the code performs the same calculations twice while in the base run the calculations are performed only once. However, the calculations that result at the end of each advancement must be the same.

The statement of the null hypothesis, Statement (1.1.3), must be modified.

H_0: The two runs produce exactly the same calculations at the end of each successful advancement (1.1.3')

Note though that this adjustment encompasses the original statement of Statement (1.1.3). It was not made originally for simplicity of exposition. With Statement (1.1.3'), the proof of Theorems 1.2.1 and 1.2.2 remains the same.

THEOREM 1.2.3: For backups, Test (1.1.5) commits no Type I Error. It has significance level, $\alpha = 0$.

PROOF: Suppose H_0 is true. Then the two code runs produce identical calculated results at the end of each successful advancement. This is true despite the fact that one run performs as many as twice as the number of calculations as the other on the timestep(s) where the false backup is forced.
As before, $X_i = 0$, for i = 1, 2, …, N and so $X = \max \{X_i \mid i = 1, 2, \dots , N\} = 0$, and it follows that $P(X=0 \mid H_0 \text{ is true}) = 1$. Following the same argument as above

$$\alpha = P(\text{Reject } H_0 \mid H_0 \text{ is true})$$
$$= P(X=1 \mid H_0 \text{ is true})$$
$$= 1 - P(X=0 \mid H_0 \text{ is true}) = 1 - 1 = 0.$$

Q.E.D.

1.3 Power of the Testing System

In order for this test to be effective, to have power approaching 1, there must be a good chance of detecting differences between two versions caused by a coding change. Similarly it must be able to detect differences in restart and backup testing. There are two important aspects to this:

- detection – finding errors in the coding tested
- coverage - percentage of coding exercised by the test set

The first requires the test to have a high probability of *detecting* differences in the calculations of two different code versions *for a given test case*. Thus, it is desirable that each d_i equals 1 where

$$d_i = P(X_i = 1 \mid \text{There is a difference in test case i}) \quad (1.3.1)$$

As explained in Section 1.1, this is reduced by improving detection of differences and increasing coverage. It is possible to guarantee detection of any difference between two code runs for any input deck, i.

STRATEGY 1: For Test (1.1.5), compare all values of all variables to the last bit at every advancement and report any difference found.

THEOREM 1.3.1: For Strategy 1, $P(X=1 \mid \text{There is a difference in test case i}) = 1$.
Strategy 1 has a 100% probability of detecting differences between two runs in the verification test set.

PROOF: Suppose there is a difference between two runs for test case i. Then at least one variable must have a different value between the two runs on some timestep. Since Strategy 1 compares all values at all timesteps, it compares those two values and reports finding the difference. By Definitions (1.1.1) and (1.1.2), $X_i = 1$ and therefore $X = 1$. Thus,

$P(X=1 \mid \text{There is a difference in test case i}) = 1$.

Q.E.D.

Of course, Strategy 1 is unnecessary and impractical. Section 2 develops a practical method and explains that it is nearly as effective as Strategy 1.

The second aspect of test power, coverage, is important to Theorem 1.3.1. Despite the thoroughness of Strategy 1, many differences will escape detection unless the coverage of the verification test set is sufficient. Theorem 1.3.1 requires a *difference <u>in test case i</u> of the verification set*.

Consider an example. Suppose H_0 is false because two versions of RELAP5-3D have coding that produces different calculations. If none of the tests exercise that coding, Test (1.1.5) with Strategy 1 will calculate that X=0 and accept H_0 even though H_0 is false. That is Type II Error,

$$\beta = P(\text{Accept } H_0 \mid H_0 \text{ is false}) > 0 \quad (1.3.2)$$

but in this example, $P(X=1 \mid \text{There is a difference in test case i}) = 0$. Thus,

$$1-\beta = 1 - P(\text{Accept } H_0 \mid H_0 \text{ is false}) \leq 1 - P(X=1 \mid \text{There is a difference in test case i}). \quad (1.3.3)$$

The power, $1 - \beta$, of Test (1.1.5) increases as the coverage of the coding increases. Ideally, when comparing two code runs that could produce differing calculations, the difference is detected by at least one of the test cases (coverage) with 100% probability. That is:

$$d = \min\{d_i \mid i = 1, 2, \ldots, N\} = 1. \quad (1.3.4)$$

As with Strategy 1, this ideal is impractical for any program as complex as RELAP5-3D, especially if a goal for the verification test set is to run in a timely manner. It is more important to cover code features in common use and those important to even one or two users. It is not important to cover features that are seldom used or have not been exercised in many years. If users report errors in intentionally omitted features, tests of those features can be added to the verification test set at that time.

Therefore, since coverage is not complete, the null hypothesis will be accepted sometimes when it is false. The power of the test can be increased by improving the coverage of the code. As a starting point, the most important code capabilities should be tested by incorporating relevant test cases in the test set. Section 3 explains the verification test set.

1.4 References

1-1 The RELAP5-3D Code Development Team, "RELAP5-3D Code Manual Volume I: Code Structure, System Models and Solution Methods," INL-EXT-98-00834-V1, Revision 4.0, Section 8.2, p 8-4, June, 2012.

1-2 The RELAP5-3D Code Development Team, "RELAP5-3D Code Manual Volume III: Developmental Assessment," INL-EXT-98-00834-V1, Revision 4.0, Section 8.2, p 8-4, June, 2012.

1-3 G. L. Mesina, "Reformulation RELAP5-3D in FORTRAN 95 and Results," Proceedings of the ASME 2010 Joint US-European Fluids Engineering Summer Meeting and 8th International Conference on Nanochannels Microchannels, and Minichannels, FEDSM2010-ICNMM2010, Montreal, Quebec, Canada, Aug 1-5, 2010.

1-4 D. L. Aumiller, G. W. Swartele, J. W. Lane, F. X. Buschman and M. J. Meholic, "Development of Verification Testing Capabilities for Safety Codes," The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, NURETH15-145, Pisa, Italy, May 12-17, 2013.

1-5 V. K Rohatgi, An Introduction to Probability Theory and Mathematical Statistics, Second Edition, ISBN-10: 0-471-34846-5, John Wiley & Sons, Inc., NY, Oct 2000.

2. VERIFICATION FILE

For effective null testing via Test (1.1.5), the probability of detecting a change that affects RELAP5-3D calculations should be close to 1.0. In other words, a test with minimal Type II error, β, is sought.

$$\beta = P(\text{Accept } H_0 \mid H_0 \text{ is false}) = P(X=0 \mid \text{There is a difference in at least one case}). \quad (2.0.1)$$

According to Theorem 1.3.1, it is possible to attain 100% detection so that $d_i = 1$. Data needed for Strategy 1 must be written on a disk file for each run and their contents compares. This approach has many drawbacks:

1. *The size of the disk files can grow without bound, based on user input.*
2. *There are serious maintenance issues, every time a new variable is introduced in the code, it must be added to the write statements.*
3. *So many writes would seriously compromise code runtime.*
4. *Despite perfect detection of differences, coverage also controls Type II error.*

The requisite disk file is unacceptable because of its unlimited size can. An upper limit on file size is mandatory to avoid overfilling disk space. Although file compression can reduce size multifold and still represent all that data with 100% accuracy, even the most powerful file compression methods cannot prevent a user from overfilling disk space with a single file.

Accepting this limitation means $d < 1$ in Equation (1.3.4). The goal is to find a reduced representation of the data that bring d_i near 1 for each input case in the test set. More importantly, the file must detect differences in the basic, or most fundamental, variables. The file onto which the data is stored is called the *verification file*.

The contents of the *verification file* are detailed in Section 2.2. Reduced representations of the most important RELAP5-3D variables are dumped to the verification file on time-step advancements specified by the user in the input file. This file is similar in many ways to the verification file developed at another national laboratory for another TH code[2-1].

With verification files, it is possible to detect differences in calculations, not only between two code versions run on the same test case, but also between a base case run and a restart run, or between a base case and a run with a forced back-up. Therefore, the verification file forms the basis for Category 1, 2, and 3 testing. These are described in Sections 3, 4, and 5 respectively.

The null hypothesis must be rejected when differences occur for coding changes that are intended to produce no changes to calculations, such as code cleanup or the addition of a new feature for which there is no input file that tests it. However, *for error corrections and improvements to code features, changes in calculations are expected and even desired*. Testing such changes should result in rejection of the null hypothesis. In such cases, the developer must examine the test cases that differ to insure that differences occur only in the proper ones.

Apart from the calculated data, key features of the verification file include: unique identification of the code that ran it; the test case it ran; and the running time of the test case. This information is useful for tracking down the source of the calculation change and creating historical records of code runs.

Section 2 develops the detection of differences for any input case.

2.1 Functional Requirements for the Verification File

The following attributes constitute functional requirements for the verification file. These can be found in the Purchase Order[2-2] for this research and development project. According to that numbering system, the attributes are summarized in Table 1.

Attribute	Description of verification file and data
1	Sums of calculated values
2	Manageable size. Less than 1 MB.
3	On/off switch for verification file
4	Specify name for verification file: default & via input
5	Specify start and end time via input
6	When on, automatically write verification data for final timestep
7	Unique identifiers for code version and computer name
8	Execution time

Table 2.1.1 Attributes of a Verification File.

Attributes fall into two categories: user control and verification file contents. User control involves activation of the verification file dumping, renaming the file from its default name, and the times at which verification dumps are made. This is detailed in Section 2.2. Additional rules for verification file operations are covered in Section 2.3. In Section 2.4, the contents of the verification file are explained in detail.

2.2 Verification File Control

Verification control attributes are summarized in Table 2.2. These represent subtasks for the work necessary to produce the requisite user controls and specifications for the verification file. In this section, the tasks are organized into chronological order by task.

Task	Attribute	Control of verification file
1	3	Runtime on/off switch for verification file – no recompilation
2	4	Specify name for verification file: default & via input
3	5	Start and end time specification via input
4	6	When on, automatically writes verification data on final timestep

Table 2.2.1 Control Tasks of a Verification File.

2.2.1 199 Card Control

Attributes 3, 5 and 6 are handled by modifying the 199 card[2-3]. This card was originally developed to provide user control of start and end time for dumping debug information, much as the 105 card does but with much greater control. The 199 card allows two keywords besides the start and end time. The first keyword indicates the major type of information to dump, such as for the DTSTEP test matrix, linear equation solver, or verification file. The second keyword indicates the action undertaken. Thus, the format for a 199 card is:

199 keyword Action T-start T-end (2.2.1)

The keyword "verify" activates the verification file, and action is "dump" or a back-up condition.

2.2.2 Attribute 3: On/Off

Previously for a 199-card, the only allowable keyword was "DTSTEP." New keywords "verify" and "noverify" have been added. The "199 verify" card turns on verification dumps while "199 noverify" turns it off. Thus, it is only possible to use "199 noverify" in an input file with multiple input cases; a later input case can turn verification off for itself and subsequent input cases of the same deck.

This modification of the 199-card implements attribute 3 of Tables 2.1 and 2.2, for if a 199-card is included in an input deck and word 1 is "verify," the verification file is active. If it does not occur, no verification file is written.

Currently, only one 199 card can occur per case of a RELAP5-3D input deck. A second 199 card is treated as a replacement if it occurs in the same input case. It is, however, possible to expand this capability in the future so that a single 199 card may contain multiple sets of "199 data" as is done elsewhere in RELAP5-3D. That would, for example, allow the writing of a verification file while running the DTSTEP Test Matrix simultaneously.

2.2.3 Attribute 5: Start/End Time

Attribute 5 is the ability, via input, to specify start and end time: T-start and T-end, where:

- T-start is the TIMESTEP on which to activate the writing of data to the verification file.
- T-end is the TIMESTEP on which to deactivate the writing of data to the verification file.

Thus if "199 verify" occurs in the input deck, a verification file dump is written at timestep T-start and every timestep thereafter until T-end.

This has been expanded.

For sensitive input models whose number of advancements change frequently when the code is modified, such as TYPPWR, it is useful to specify a starting time based on cumulative time, rather than the number of attempted advancements. Therefore, T-start can also specify a floating point time on which to begin verification dumps. This means *T-start can be an integer or real*.

T-end is *always an integer*.

- When T-start is *floating point*, T-end is interpreted as the number of timesteps, counting the timestep at T-start, on which to make verification dumps.
  - For example "199 verify dump 1.0 1" will dump at times 1.0 sec and on the final step.
  - Similarly, "199 verify dump 1.0 2" will dump at times: 1.0, 1.0+dt, and the final step.
- When T-start is *integer*, T-end is interpreted as advancement (NCOUNT) on which to make the last dump before the automatic final dump.
  - For example "199 verify dump 100 102" will dump on advancements 100, 101, 102 and on the final step.

This has been further expanded.

To mimic the restart notation for specifying the final timestep with a negative one, the user may specify T-end = -1. In that case, verification dumps will be made from T-start to the end of the transient unless the verification file gets too large. With this mechanism, the verification file allows approximately 800 verification dumps for a single input case as currently configured; see Section 2.4, Equation (2.4.2).

2.2.4 Attribute 4: Input Naming of Verification File

Attribute 4 is the ability to name the verification file. The default name is "verify." If an output file has been specified through command line or input deck, the default verify-file name has the same base file name, but has the file extension "vrf."

The user can override these two default names via the command line option "–R." If the 199 card is present and calls for the verification file, and the "-R" command line flag is used, then its argument is the verification filename. This is summarized in Table 3.

In the implementation, variable VERIFL holds the Fortran unit number for I/O statements (open, close, read and write). In UFILSMOD, its value is set to 18, replacing the old direct access restart plot file that was eliminated long ago. In the UFILFMOD it occupies position 15 in the FILSCH array. The coding that implements Table 3 is a new internal subroutine in GNINIT1 called NameAnOutputFile.

199 card	Word 1	Restart output file	-R argument	Verify File	Verify File Name
Absent	N/A	any	any	No	N/A
Present	Verify	none	none	Yes	verify
Present	Verify	restname.out	none	Yes	restname.vrf
Present	Verify	restname.out	verifyfilenm	Yes	verifyfilenm
Present	Anything else	any	any	No	N/As

Table 2.2.3. Naming the Verification File.

2.2.5 Attribute 6: Dump on Final Step

Attribute 6 is the writing of a verification dump on the final timestep of a transient run whenever the verification file is activated, *and the code does not abort early*.

The code was modified to implement this requirement by always calling the subroutine VERFSUM, displayed in Section 7, in DTSTEP whenever VERFACTION > 0. VERFSUM checks for final timestep and gathers all verification data if it is. In fact, VERFSUM checks if either the variable DONE is non-zero or variable FAIL is true. If either condition holds, it writes the final verification dump at that timestep as required. Since it is called after DTSTEP sets these conditions, VERFSUM catches graceful shut-down situations (termination of runs *not* caused by the code aborting) in the transient and ensures that the final verification dump is made.

2.3 Verification File Operations beyond Original Workscope

Rules for handling files and input deck cases that apply to the verification file are listed here. These additional issues and features were not addressed in the original project workscope.

1. The verification file can be named with the -R command line option.
2. An allowance to overwrite the verification file has been programmed. The normal 'Zen of RELAP' disallows the overwriting of an important file, and quits with an error message.
3. In decks with multiple cases, data for all cases is written. It goes on the same file while verification is active. On the verification file, the cases are identified by their case number and title.
4. In decks with multiple cases, the verification file can be deactivated by a "199 noverify" card. The verification file remains open until a "199 noverify" card is encountered in an input case. Once closed, the verification file remains closed for subsequent cases. If "199 verify" card is not present in the first case of an input deck with multiple cases, but occurs in a later case, no output to the verification file occurs for cases that precedes the 199-card; however the verification file is activated for the case in which the card occurs and remains active for all remaining cases unless deactivated in a subsequent case.
5. In decks with multiple cases, the verification file may be reopened in a case subsequent to a case with a "199 noverify" card through the use of a subsequent "199 verify" card. Thus a user may turn verification on and off among the cases as often as desired. The output goes to the same verification file for all cases of the same input deck.
6. The timesteps dumped on the verification file remain the same from case to case unless changed with a replacement "199 verify" card or turned off.
7. The key value, -1, may replace the end time (or end advancement number) and means make verification dumps until the final timestep.

2.4 Verification File Contents

Verification content attributes are summarized in Table 2.4.1. These represent subtasks for the work necessary to produce the requisite contents of the verification file. In this section, the tasks are organized into chronological order by task.

Task	Attribute	Content Description
5	7	Unique identifiers for code version and computer name
6	1	Sums of calculated values
7	8	Execution time (CPU time)
8	2	Manageable size of less than 1 MB.

Table 2.4.1. Content Attributes of a Verification File.

2.4.1 Format of the Verification File

The verification file is written in human-readable ASCII format with textual annotations. It has a maximum size of 1 MB and has three sections of information, namely the header, body, and footer.

The header contains the unique identifier for the RELAP5-3D program that produced the verification file. It also contains the date and time the run was made.

The footer section contains the execution time (Attribute 8 in Table 2.4.1) for the entire problem.

The verification file's body section contains the information dumps for each input case that is active writing to the verification file. Within the section of a given input case, the data dumps for the user-requested advancements are placed; ending with the dump for the final timestep of the case.

The data for a given input case begins with the input case number and title of the input case. After a blank line, the advancement number and number of the verification data dump within the input case is recorded. Next follows annotated L_1-norm of floating point and integer arrays. The actual arrays summed are given in Table 2.4.2 of Section 2.4.3. Integers are written in eight-byte format. Floating point values are written in two formats. The first is 1pE24.16 for human readability, the second is HEXADECIMAL, z32. The reason for HEX is that floating point representations, even at 16 places, may not reveal a difference in the last bit, but hexadecimal will.

2.4.2 Code and Computer Identifiers

Attribute 7 is the unique identifier information for the RELAP5-3D program. There are two pieces of data necessary to uniquely specify the RELAP5-3D run:

1. Code version number – The RELAP5-3D executable always had this hard coded.
2. Time program was built – Main program was modified to capture this via pre-compiler directives.

These are sufficient to uniquely identify the executable program that was used in the analysis, even when multiple versions of the code can exist with the same internal version number. The computer identifier is obtained via FORTRAN intrinsic[2-4] `get_environment_variable` to access the value of HOSTNAME on Linux platforms, and firstboot on Windows-7 platforms. Note that firstboot only tells the date and time of the machine's first boot-up, but that is virtually unique. Version, build-time, and computer identifier go into the header.

2.4.3 Execution Time

Attribute 8 is information about the execution time of the simulation. The value reported is the CPU execution time the RELAP5-3D reports on the printed output file. For a given input case, RELAP5-3D records the timing measured from the beginning of input processing to the end of the transient or steady state calculation in s_stscpu. Where there are multiple input cases, the value reported is the sum of all input case timings. Execution time goes into the footer.

2.4.4 Norms of Calculated Quantities

Attribute 1 of Tables 2.1.1 and 2.2.1 is a set of sums of independent values. These sums are the quantities that reveal changes in code calculations. The quantities that are summed are listed in Table 2.4.2. The Table includes the symbol used by the RELAP5-3D manuals for the quantity and the area of the calculations to which the quantity applies.

	Notation	**Quantity**	**Symbol in Manual**	**Identifier on the file**	**Area**
1	P	Pressure	p	P	TH
2	Uf	Liquid internal energy	u_f	Uf	TH
3	Ug	Gas internal energy	u_g	Ug	TH
4	VOIDg	Void fraction of gas	α_g	VOIDg	TH
5	QUALa	Noncondensable quality	X_a	QUALa	TH
6	Boron	Density of boron	$\boldsymbol{\rho}_b$	Boron	TH
7	Vf	Liquid velocity	V_f	Vf	TH
8	Vg	Gas velocity	V_g	Vg	TH
9	RHSth	RHS of Δp linear system	b	RHSth	TH
10	SOLth	Pressure drop / velocities	Δp / (V_f, V_g)	SOLth	TH
11	Error	Errors	ε		Advancement
12	Temp	Heat Structure Temperature	T	Temp	Heat Transfer
13	Flux	Neutron flux	φ	Flux	Neutron Kinetics
14	dtsum	Timesteps sum	Δt, Δt_{kin}	dtsum	Advancement
15	Trips	Trips	T_r	Trips	Trips
16	Cntrl	Control system value	Y	Cntrl	Controls
17	Rdc	Reductions	N/A		Advancement
18	Rpt	Repeats	N/A		Advancement

Table 2.4.2. Quantities on Verification File.

The L_1-norm, or sum of absolute values of array entries, is calculated for each of the 18 quantities in quadruple precision. For a vector $z = (z_1, z_2, \ldots z_n)$ the L_1 norm is:

$$\|z\|_1 = \sum_i |z_i| \tag{2.4.1}$$

The L_1 norm prevents the possibility of cancellation between positives and negatives accidentally producing the same sum when the arrays are different. Most of the quantities in Table 2.4.2 are primary variables in the governing equations. The remainder, such as errors and repeats, are measures of code performance. The quantities in Table 2.4.2 go into the body of the verification file.

2.4.5 Verification File of Manageable Size

Attribute 2 is satisfied by the small number of values written to the verification file. Since all of the quantities are sums, the total number of values can be calculated as follows:

OUTPUT	BYTES
Header	
Code / Version / Computer	≤ 83
Compile Time	35
Run Date/Time	43
Blank line	1
Subtotal	≤ 98
Body – Case Title	
Case Number and Title	≤ 90
Blank Line	1
Subtotal	≤ 91
Body – Advancement	
Dump & advancements numbers	52 each
18 floating pt. sums, 62 bytes each	1132 each
2 lines of integer sums, 60 bytes each	120 each
Blank Line	1
Subtotal	1305
Footer	
CPU time & size	46
Subtotal	46
Body + Footer	1351

The formula for the upper limit on the size of the file is:

$$S(n, t_1, \dots, t_n) = (98 + 46) + 91n + 1305(\textstyle\sum_{i=1}^{n} t_i). \quad (2.4.2)$$

where n is the number of input cases within the input file, and t_i is the number of timesteps requested for timestep dump i. This is an upper limit because the title may be fewer than 80 characters and the code can abort before completing the run. When t is the number of timesteps per case for every case, the limit for the size of the file reduces to:

$$S(n, t) = 144 + 91n + 1305nt \quad (2.4.3)$$

The size of a single advancement dump slightly exceeds 1 KB. Thus for a single input case, over 800 verification dumps can be made. Over 750 input cases, each with a single dump on the final timestep, could fit the 1 MB limit. However, since the user controls the number of cases and dumps, safeguards have been coded to limit the verification file size to remain under the 1 MB upper limit.

W*hen the file grows to within one dump of the upper limit, an error message is written both on the screen and the printed output file*. This leaves sufficient space for the required dump on the final step of that input case, therefore, no other verification dumps are made for the current input case until the final timestep. Thereafter, the verification file is closed and cannot be reopened.

2.4.6 Sample Verification File

The verification file for Edward's Pipe problem edhtrk.i activated by card "199 verify dump 0.1 1" is presented in Figure 2.4.6. It was produced by RELAP5-3D version 4.1.3 on computer steelers.inl.gov by a code compiled on Aug 13, 2013. This information is in the header. The body displays case title, dumps by advancement and time, the automatic dump at the end, and the sums. The footer has the run time.

Figure 2.4.6. Verification File with 2 Cases for Edward's Pipe

```
RELAP5-3D/Ver:4.1.3   steelers.inl.gov
Time compiled: Aug 13 2013 13:29:15
Date and Time of run: 13/08/14     15:04:49

Case  1  edward's pipe problem base case with extras

Dump     1    Advancement=     109 time=  1.0000E-01
P=      4.9365983737086219E+07 401878A1EFDE58D75B00000000000000
Uf=     1.9649507480408072E+07 40172BD3E37AFC05FEC0000000000000
Ug=     5.4520489485535964E+07 40189FF554BE260AE000000000000000
VOIDg=  7.0158488970410998E+00 4001C103AB179E074A00000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.0448213290728118E+02 400698F6DA1FDA3236D4000000000000
Vg=     2.3165076689908255E+02 4006CF4D3151A9C1FEC1000000000000
RHSth=  0.0000000000000000E+00                                 0
SOLth=  5.2542461771631456E+04 400E9A7CEC6D54CEA4E0000000000000
Error= -8.5282658356481664E-05 BFF165B38EA0ADAA2000000000000000
Temp=   1.1047897158084513E+05 400FAF8EF8B985B33F57500000000000
Flux=   6.4046362410846550E+10 4022DD2EBDE55B16F000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -3.9020138535691576E+00 C000F37530A0CF29DB80000000000000
Cntrl=  3.7065329809843512E+06 4014C47527D90E52D0F595356B020000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

Dump     2    Advancement=     509 time=  5.0000E-01
P=      1.1610017826711973E+07 4016624F43A746CAAC00000000000000
Uf=     1.3706563288757732E+07 4016A24A8693D80DB180000000000000
Ug=     5.3792556235069888E+07 40189A67961E16C52400000000000000
VOIDg=  2.0127747744316551E+01 4003420B4137FFA34180000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.8891214895206032E+02 400720E98297FE2E04B8000000000000
Vg=     9.1675057057565303E+02 4008CA6012B255E284C0000000000000
RHSth=  4.2453960924539154E+07 401843E5E476574C8C12980000000000
SOLth=  1.6144078316381101E+05 40103B50643EB635D838000000000000
Error= -9.9606881069212402E-05 BFF1A1C812FC4B5E8000000000000000
Temp=   1.0939814425864978E+05 400FAB5624EE2286FA5FD00000000000
Flux=   2.7820142401306227E+07 4017A8806E66BC014000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -1.6980010000000000E+00 BFFFB2B0318B934698000000000000000
Cntrl=  8.6399604127190748E+05 4012A5DF815219769C2F2BB3AB200000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

CPU Time=  3.6094499999999996E-01 size    2764
```

2.5 Analysis of Verification File Sufficiency

This section examines whether or not the quantities written to the verification file are sufficient for detecting differences between runs of the same test case on two different code versions. The following definitions apply to the discussion:

- Primary variable – one of the fundamental variables identified in the governing equations in the manual.
- Secondary variable – not part of the governing equations, calculated from primary variables, feedback into the primary variables.
- Tertiary variable – calculated for display only and does not feedback to primary or secondary variables.

It will be shown that all errors in calculation of primary and secondary variables are caught by the verification file and testing procedure (1.1.5), except for those lost to quadruple precision summing. Errors in tertiary variables cannot be caught by the verification file. Tertiary variables can be important for simulator application, for example an alarm trigger. This may prove important for future development.

2.5.1 Primary Variables of the TH equations, Rows 1-8

Pressure, density, void, both internal energies, noncondensable quality, and both velocities are primary variables. The user has an idea of the right order of magnitude of these norms, so it gives a more intuitive understanding of the situation than the norm of the TH system RHS or Δp solution array would (in cases where the semi-implicit time advancement scheme is in use). L_1-norms of these 8 arrays are essential.

Consider what would happen if any of these quantities were calculated incorrectly in one of the two runs being compared by Test (1.1.5). Either on the same timestep or the next, the secondary quantities calculated from them would also have incorrect values. This eliminates the need to place secondary variables on the verification file, except to verify the coding that calculates them. With few exceptions, the differences will persist and often grow as the incorrect values propagate differences throughout the solution. These differences will be displayed on the next verification dump.

The most obvious exception would be steady state TH conditions that strongly smooth out the error before the next dump can be made. It is advisable to have an intermediate dump for input models such as this.

Now consider the need of placing secondary variables on the verification file. If coding that computes any of them is incorrect, the errors will propagate into other secondary variables, affect the construction of the discrete system of governing equations on the next timestep, and alter the calculation of the primary variables. With the exception noted, this change in primary variables will result in altered values on rows 1-8 at the next dump. Little is gained by adding secondary variables to the dumps, other than helping to recognize where an error occurs. However adding secondary increases the size of the verification file, so it is not done. Debugging with the verification file is covered in Section 2.6

2.5.2 Heat Structure Temperatures and Fluxes, Rows 12-13

The temperatures are summed across all mesh points. With a similar argument as presented for TH, if temperature or flux were calculated incorrectly, derived quantities would also be calculated incorrectly. Moreover, the user knows what kinds of values to expect.

Until it is shown insufficient, only the L_1-norms of these two quantities will be dumped on the verification file.

2.5.3 TH Equation RHS and Solution Vector Norms, Rows 9-10

The right hand side of the TH equations could be considered a primary variable, but because all the actual primary variables are normed and displayed, these are redundant. Moreover, the user has no feel for the values of the right hand side array as they do for the primary variables. For the semi-implicit timestep methods the control volume pressure drops, Δp, are the most fundamental of all variables because all other primary TH variables are back-solved from them. For nearly implicit however, the solution is velocity-based and the pressure drop sum winds up as zero. For these reasons, these are not considered essential, but are calculated and presented. They can be ignored for comparison of state points. They are primarily useful for debugging as is shown in Section 2.6.

2.5.4 Sum of trip and control variables, Rows 15-16

Trip and control quantities can change based on time, tables, component action. The result of trips and controls may be to activate an alarm for plant operators, a tertiary variable. Trip and control quantities are primary variables that do not completely depend on the TH, Temperature and Flux variables and may not affect them in return. Therefore their L1-norms are calculated and dumped on the verification file.

2.5.5 Timestep and Error sums, Rows 11, 14

The timestep information indicates to the user what the rate of progress through the transient is.

The mass residual ratio and error estimate are critical to timestep/advancement strategy and for evaluating the closeness of the approximations

2.5.6 Sum of key output-only quantities, Rows 17-18

None of these should differ between comparable runs. They are completely dependent on the primary variables and are only included to aid in debugging. They are:

- Sum of the reasons for time-step reductions including: noncondensable quality, extrapolation, mass error, fluid property and courant limit violations.
- Sums of reasons for backups.

No other quantities, secondary, tertiary, or temporary are dumped to the verification file. As noted, errors in secondary quantities eventually affect primary quantities. The quantities in (1) – (4) cover the calculations of the primary physics at the 100% level, *if computer arithmetic were perfect*. However computer arithmetic is not perfect. Issues with the integer sums (5) and floating point are addressed next.

The integer output quantities count the number of times something, such as a particular type of back-up condition, happens at a given location, such as a control volume. The sum of those happenings across the entire vector of locations may be the same even though the locations where they occurred in two runs that are being compared differ.

Also, the standard 8-byte floating point word has about 14 decimal digits of accuracy. If one member of the array is 15 orders of magnitude smaller than another, it contributes nothing to the sum. A difference in this member of the array in two corresponding runs results in identical numbers being written to the verification file. To somewhat overcome these round-off error issues, floating point sums are accumulated in quadruple or 16-byte precision. This gains at least 10 decimal places. Moreover, these sums are written in Z32 Hexadecimal format to represent every one of the 128 bits in the sum.

The L_1-norm, which is the sum of the absolute values of the array values, was chosen to eliminate the possibility that two different arrays could have the same sum due to arithmetic cancellation. This helps to reduce the Type II Error, Equation (1.1.9) and increase the power, Equation (1.1.10) of Test (1.1.5).

Comparison of the verification dump file analysis with Strategy 1 shows the following deficiencies in their detection capabilities.

1. While it is true that L_1-norm values can be equal for two different arrays, all primary variables are interdependent. It is most unlikely that the L1-norms of all these arrays will be equal while all the arrays differ, except in trivial (contrived) cases. In real plant models, the chance is nearly 0%.
2. While the 24 decimal place accuracy of quadruple precision sum can be equal for two different arrays, the values lost to the sum are vanishingly small. Except for computer noise in uninitialized quantities, this should never happen in real plant models. The chance is nearly 0%.
3. The absence of secondary variables causes no loss of detection unless the transient ends before the error in a secondary variable manifests itself in a primary variable, and that normally occurs on the next timestep.
4. Errors in tertiary and temporary variables, except those listed in Section 2.5.6 above, will not be detected.

Other than those listed in Section 2.5.6 above, the verification file cannot detect errors in output only quantities and therefore cannot guarantee that code output is correct. For example, if a change resulted in the incorrect reporting of a flow regime as bubbly rather than pure liquid, the error would affect neither primary variables nor any other quantity in the dumped information, and so the verification file cannot help with detecting or debugging that. This demonstrates that the verification file does not detect differences with 100% effectiveness. It does commit Type II error. However, this is a small defect in detection.

This analysis demonstrates that differences between two runs in important calculated quantities for non-trivial models will be detected nearly 100% of the time.

2.6 Debugging with the Verification File

The verification file provides useful information for locating the source of change. If a difference between two code versions running the same input file is discovered, and no change to the source code was expected to cause it, the difference is an error. To find the error the follow steps can be taken.

1. Turn on verification dumps at every timestep in the vicinity of the time the error is suspected to occur. Repeat this until the first timestep with a difference is found.
2. For the timestep of the first difference and its preceding step, compare the solution vector and right hand side norms.
    a. If the RHS sums are the *same*, but the solution vector sums *differ*, then the solver has an error.
    b. If *both* RHS and solution vector sums *differ* between versions, then non-solver routines committed an error.
3. For non-solver errors, if the information in Section 2.5 items (4) and (5) differ between the runs, then the error occurs in the subroutines called after the solver routines on that timestep.
    a. If those data are the same, then the error occurs on the next timestep in subroutines called before the solver routines.

This is summarized in Table 2.6.1.

RHS sum	**Solution sum**	**Output-only**	**Location of error**
Same	same	N/A	no error
Same	differ	N/A	solver
Differ	same	N/A	solver
Differ	differ	differ	this advancement after solver
Differ	differ	same	next advancement before solver

Table 2.6.1. Debugging Use of the Verification File.

2.7 References

2-1 D. L. Aumiller, G. W. Swartele, J. W. Lane, F. X. Buschman and M. J. Meholic, “Development of Verification Testing Capabilities for Safety Codes,” The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, NURETH15-145, Pisa, Italy, May 12-17, 2013.

2-2 C. Gross, “Restart and Backup Testing,” Purchase Order Number 7001655, Amendment 3, BMPC, Sep 26, 2012.

2-3 G. L. Mesina and D. L. Aumiller, " DTSTEP: Development of an Integer Time Step Algorithm," RELAP5 International Users Seminar, Park City, UT, Aug 10-13, 2010.

2-4 J. C. Adams, et al, The Fortran 2003 Handbook, The Complete Syntax, Features, and Procedures, ISBN 978-1-84628-378-9, Library of Congress 2008934286, Springer, springer.com, October 31, 2008.

3. NULL TESTING

Null testing, or Category 1 testing, makes use of the verification file specified in Section 2 to detect differences between two RELAP5-3D versions applied to the same input. As presented previously, there are two fundamental aspects controlling the Type II error of Test (1.1.5), namely, the detection of differences for a given input case and the coverage of the program provided by the test cases of the Test suite.

In Section 2.5, it was shown that the verification file provides excellent detection of differences between two code versions running the same input file. In Section 3, the coverage of the RELAP5-3D program is examined.

Typically, coverage analysis examines the lines, functions and subprograms of a code. Instead, the features of the code are considered. The features include the most commonly used features in RELAP5-3D as well as certain other important features. Those features deemed to be of lesser importance, because either they are seldom exercised or are not important for modeling nuclear power plants, are not included in the features tested by the suite. However, the suite can be expanded to include testing of more features in the future.

Once the features of RELAP5-3D were identified, a collection of input problems was developed to test them. Much of this was done by collecting relevant extant test cases. These input decks were modified to test additional important code features.

Section 3.1 contains the functional requirements for the Category 1 or null testing. Section 3.2 lists the tested code features in tabular form. There are over 100 and they are documented in the RELAP5-3D manuals, so they are not expounded upon in the Section. The input models of the test suite are described in Section 3.3. The automated means is discussed in Section 3.4. Section 3.5 presents the verifications run on two identical machines for comparison. Section 3.6 supplies references.

3.1 Functional Requirements

The functional requirements or attributes for null, or Category 1, testing can be summarized as follows[2-2, 3-16]:

1. A collection of input problems that test important features of RELAP5-3D.
   - The collection of input problems that test the most commonly used features in RELAP5-3D must be developed.
   - A description of the test features that are included in the test suite must be provided.
   - The list of problems and the descriptions of features must be approved by the purchaser.
   - The list shall also include all features that are known to be absent from the test suite.
2. A simple method for performing null testing:
   - An easy and automated means to perform null testing must be provided.
   - This must provide the ability to test different versions of the code or to test the results on different platforms.
   - This method must provide an unambiguous statement concerning the success of lack of success of the null testing.

When two runs produced by different code versions are compared, there may be differences due to code development, code maintenance, code error corrections, differences in compiler levels and options, and other operating system and hardware differences. Therefore differences do not necessarily indicate an error has been discovered. However, care must be taken to ensure that the compiler, operating system and hardware are the same before any comparison is made.

3.2 Features and Cases Matrix

A review of the code's capabilities and features that are commonly used in performing reactor and associated system simulations was performed. The important categories of code features and models included hydrodynamic components, volume and junction options, heat structure types, correlations, boundary conditions, trips, tables, control variables, reactor kinetics, Appendix K, and user choices that affect the way the code operates. Among user choices are time advancement scheme, solver, card 1 options and many others.

3.2.1 RELAP5-3D Features

Groups of features were expanded to list individual ones. For example, a valve is a hydrodynamic component and six kinds of valves are listed. Altogether *112 features were selected for testing*. This represents all code features commonly used to model nuclear power plants, and numerous uncommon ones.

Rarely used components, models, and features were excluded from the verification test suite.

- The mechanistic General Electric separator component is not tested but does appear in the matrix.
- Several rarely used heat transfer packages and options were excluded from the matrix.
- Card 1 options, debug and print control options, minor edits, expanded minor edits, and interactive variables are not systematically tested, though specific Card 1 options are tested.
- Not all fluids are tested.

However Test cases were developed for all the code's different interpolators as exercised by H_2O, D_2O, H_2O-New, and "new" ATHENA fluids, such as new helium. Moreover, the principal timestep level-of-implicitness options (tt = 3, 7, 11, and 15) are specifically exercised for a few of the cases in the null test set. If desired, these options can readily be tested with the entire test suite using recently developed methods[3-13].

3.2.2 Input Decks

Based on testing these features, the verification test suite of input decks has been developed for verification restart and backup testing. In Section 3, the emphasis is on Category 1 or null testing, however the same cases are used for the other two testing categories.

Input decks were selected for the test suite based on run speed and ability to test given features. It is not necessary to run long transients to test most features, many can be tested in short transients. Therefore with the exception of TYP12002.i, each test case in the suite runs requires only a few CPU seconds. The entire suite of calculations can be performed in a few minutes on a single processor. Even though it requires much more CPU time than the other test cases, TYP12002.i is included in the test suite because it exercises numerous two-phase models and is known to be very sensitive to minor coding changes during the latter stage of the transient.

Many of the decks contain multiple cases, as listed in Section 3.2, Table 3.2.1. Thus at present, the verification test suite contains 43 input decks comprised of 125 input cases. More RELAP5-3D simulations are performed if restart or backup testing (Sections 4 and 5) are also run.

3.2.3 Tabulation of the Verification Suite

The verification test suite is comprised of the input decks that test the selected features. The deck that tests a specific feature is recorded in the Features-Cases Matrix of Tables 3.1.1 – 3.1.6 by a mark under the input deck indicates in the rows of the features tested. About half of the entries in these six tables are generated by a script, the rest are marked by hand. Rows hand-marked typically have only one "X," even though more decks may test the feature. Having a single case that tests a given feature or model is assumed to be sufficient. *Therefore for a given feature, not every problem that tests it is marked* in the Tables.

In the "Features-Cases" Matrix, two indicator columns follow the list of features. An "X" in Column two indicates the feature is tested by at least one member of the suite of cases, while a blank indicates no testing. An "X" in Column three indicates feature is restarted, while a blank indicates no restart.

Two important assumptions were made in the construction of the verification test suite:

1. The verification suite does not exercise all available user options for each code feature. Basic capabilities are demonstrated, but complete code coverage of individual features is not attempted.
2. If a feature is tested in one kind of component, it need not be tested in another.
   a. For example, if Water Packing is tested in a PIPE component, it need not also be tested in a SNGLVOL or BRANCH.

Since both hydrodynamic components and water packing are code features, without assumption two, the Features-Decks matrix would need to be multi-dimensional. Moreover, without these assumptions, the number of test cases necessary would be prohibitive.

Table 3.1.1. Features-Cases Matrix – Hydrodynamic Components

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boronm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	eccmix.i	edhtrkm.i	eflag.i	enclss.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l3lacc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	regime.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsamppm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Hydro-Component																																													
SNGLVOL	X	X	X							X			X			X						X	X	X					X						X	X					X	X	X		X
TMDPVOL	X	X	X	X		X	X	X		X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
SNGLJUN	X	X	X			X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	X	X	X		X	X	X	X			X	X	X	X	X	X
TMDPJUN	X	X	X			X		X		X	X			X	X	X	X		X	X	X	X	X	X		X			X		X	X	X	X	X	X	X	X			X	X			
PIPE	X	X			X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X		X	X	X		X	X			X	X	X			X	X	X	X	X	X	X	X	X	X	
ANNULUS	X	X																						X											X	X						X	X		
PRIZER	X	X																								X																			
BRANCH	X	X								X											X			X						X	X				X	X					X	X		
SEPARATR	X	X																						X											X	X					X	X		
Black box	X	X																																								X			
GE																																													
JETMIXER	X	X																					X																						
TURBINE	X	X																																							X				
FWHTR	X	X														X																														
ECCMIX	X	X									X																																			
VALVE	X	X														X								X			X								X	X					X	X	X	X	
CHKVLV	X	X																																										X	
TRPVLV	X	X																						X											X	X					X	X	X	
INRVLV	X	X																									X																X	
MTRVLV	X	X																						X											X	X					X	X	X	
SRVVLV	X	X														X																												X	
RLFVLV																																													
PUMP	X	X	X																					X											X	X					X	X		
CPRSSR	X	X																											X															
MTPLJUN	X	X		X																								X					X	X									
ACCUM	X	X																						X	X										X	X					X	X		
MULTID	X	X		X																								X					X	X									
SNGLFW																																													
MTPLFW																																													

Table 3.1.2. Features-Cases Matrix – Component Control & Specification

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boronm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	eccmix.i	edhtrkm.i	eflag.i	enclss.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l31acc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	regime.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsamppm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Variable volume																																														X
Volume Flags																																														
t - thermal stratification	X	X																								X																			
l - mixture level	X	X																				X				X																			
p - water packing	X	X																									X																		
v - vertical stratification	X	X																								X																			
b - bundle	X	X												X						X										X	X									X					
f - wall friction	X	X													X																														
e – equilibrium	X	X																		X																									
Wall friction options																																													
Turbulent friction	X	X													X																														
Laminar friction	X	X													X																														
Shape factor	X	X													X																														
Viscosity ratio	X	X													X																														
User defined	X	X													X																														
Frictionless	X	X													X																														
Junction Flags																																													
j - jet junction	X	X																				X																							
e - modified PV	X	X											X																																
f - CCFL	X	X								X														X																				
Wallis	X	X								X																																		
Kutataledze	X	X								X																																		
Bankoff	X	X								X																																		
v - HSE	X	X														X		X																										
Top offtake	X	X																X																										
Bottom offtake	X	X																X																										
Side offtake	X	X																X																										
c - choking	X	X					X																																				
Sub-cooled	X	X					X																																				
Two phase	X	X					X																																				
Super-heated	X	X					X																																				
a - abrupt area	X	X	X				X	X		X	X		X		X	X		X	X	X	X			X		X		X		X	X	X			X	X	X					X	X	X	
h - homogeneous	X	X					X			X																						X									X				
s - momentum flux	X	X													X			X						X						X											X				
Junction form loss																																													
Constant	X	X													X																														
Reynolds dependent	X	X													X																														
Abrupt area change	X	X													X																														
Flow regimes																																													
Horizontal	X	X																														X												
Vertical pre-CHF	X	X																														X												
Vertical post-CHF	X	X																														X												
High mixing	X	X	X																																										
ECC mixer	X	X									X																																		
Drift flux models	X	X							X																																				

Table 3.1.3. Features-Cases Matrix – Heat Transfer Specification

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boronm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	ecmix.i	edhtrkm.i	eflag.i	enclss.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l31acc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	regime.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsamppm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Type Heat Structure																																													
Rectangular	X	X																																		X								
Cylindrical	X	X						X																																				
Spherical	X	X																																			X							
Heat transfer modes																																												
Forced convection	X	X																		X																								
Nucleate boiling	X	X																		X																								
Condensation	X	X																		X																								
Film boiling	X	X																		X																								
Transition boiling	X	X																		X																								
Reflood heat transfer	X	X																																						X				
2D heat conduction	X	X																																						X				
Heat structure BC types																																												
Adiabatic	X	X																																							X			
Convective	X	X																																							X			
Wall temperature	X	X													X																													
Heat flux (table)	X	X																	X																								
Heat flux (control var.)	X	X																	X																								
HTC vs. time	X	X																	X																								
HTC vs. Temp	X	X																	X																								
Alternate coupling	X	X																		X																								
Heat structure heat source options																																												
Radial	X	X																																	X								
Table	X	X																																	X								
Control variable	X	X																																	X								
Point kinetics	X	X																																	X								
Nodal kinetics	X	X																																X										
Gap conductance model																																												
Metal-Water																																												
Rectangular	X	X																																		X								
Cylindrical	X	X						X																																				
Spherical	X	X																																			X							
Material Prop																																												
Built in	X	X										X																																
Input	X	X										X																														X		
Function	X	X										X																																
Enclosure																																												
Conduction	X	X												X			X																											
Radiation	X	X															X																											

Table 3.1.4. Features-Cases Matrix – Tables and Kinetics

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boronm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	eccmix.i	edhtrkm.i	eflag.i	enclss.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l31acc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	regime.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsampnm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Radionuclide transport	X	X																																	X	X									
Reactor kinetics																																													
Point	X	X										X																								X						X			
SEPARABL	X	X										X																								X									
TABLE3																																													
TABLE4																																													
TABLE3A																																													
TABLE4A																																													
Scram (table)	X	X										X																																	
Scram (control var.)																																													
Power history	X	X										X																																	
Nodal	X	X																																	X									
RAMONA																																													
HWR																																													
GEN	X	X																																	X									
Control Rod	X	X																																	X									
Decay Heat																																													
NO-GAMMA	X	X			X																																							
GAMMA	X	X			X																																							
GAMMA-AC	X	X			X																														X	X								
ANS73	X	X			X																																							
ANS79-1	X	X			X																																							
ANS79-3	X	X			X																														X	X								
ANS94-1	X	X			X																																							
ANS94-4	X	X			X																																							
ANS05-1	X	X			X																																							
ANS05-4	X	X			X																																							
G factor	X	X			X																																							
Alternate fluids	X	X																		X	X								X										X			X	X		
Noncondensable	X	X						X		X	X	X		X				X		X	X			X	X				X	X		X	X	X	X	X	X	X		X		X	X		
Valve open and close	X	X																																									X	
Boron tracking	X	X				X																		X																				

Table 3.1.5. Features-Cases Matrix – Trips and Controls

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boxnm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	eccmix.i	edhtrkm.i	eflag.i	enclss.i	regime.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l31acc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsampmm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Trips	X	X	X							X		X									X		X		X	X	X				X					X	X				X	X	X	X	X	X
Control variables																																														
SUM	X	X	X					X		X		X			X	X	X		X	X		X			X	X				X		X			X	X	X	X			X	X	X	X	X
MULT	X	X						X		X		X			X	X			X	X		X			X					X					X	X		X			X				
DIV	X	X						X		X		X			X	X			X	X		X			X	X				X							X	X			X				
DIFFRENI	X	X										X																																		
DIFFREND	X	X										X														X																				
INTEGRAL	X	X						X		X		X													X										X		X	X				X			
DELAY	X	X								X		X																																		
FUNCTION	X	X								X		X			X															X					X								X		
STDFNCTN	X	X						X		X		X			X				X						X													X	X					X		
ABS	X	X								X		X			X				X																			X						X		
SQRT	X	X						X		X		X			X																				X		X	X				X			
EXP	X	X						X				X																									X	X							
LOG	X	X										X																																		
SIN	X	X										X			X																														
COS	X	X										X			X																														
TAN	X	X										X																																		
ATAN	X	X										X																																		
MIN	X	X										X			X																														
MAX	X	X						X				X			X										X												X	X							
TRIPUNIT	X	X								X		X													X					X					X							X			
TRIPDLAY	X	X										X																																		
POWERI	X	X										X																																		
POWERR	X	X										X			X											X																				
PROP-INT	X	X										X					X																												X	
LAG	X	X										X																																		
LEAD-LAG	X	X										X																																		
CONSTANT	X	X						X				X			X	X				X															X	X	X	X				X			
SHAFT	X	X																												X										X					
PUMPCTL	X	X										X																																		
STEAMCTL	X	X										X																																		
FEEDCTL	X	X										X																																		
INVKIN	X	X										X																																		

Table 3.1.6. Features-Cases Matrix – Code Operation Control & Misc.

Features	Present	Restart	2phspump.i	3dflow.i	ans.i	boronm.i	crit.i	cyl3.i	Drift N/A	duklerm.i	eccmix.i	edhtrkm.i	eflag.i	enclss.i	fric.i	fwhtr.i	gota27.i	hse.i	httable.i	httest.i	hxco2m.i	jetjun.i	jetpmpm.i	l2-5-emA.i	l31acc.i	neptunus20m.i	pack.i	pitch.i	radialm.i	rcpr.i	refbunm.i	reflecht.i	regime.i	rigidbodym.i	rthetam.i	rtsampnm.i	rtsamppm.i	slab3.i	sphere3.i	state.i	todcnd.i	turbine9.i	typ12002.i	typ_kindt.i	valve.i	varvol2.i
	#	#	1	2	3	4	5	6		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Tables																																													
POWER	X							X																																				
Temperature																			X																										
HTRNRATE	X	X																	X																					X					
HTC-T	X	X										X		X									X																						
HTC-TEMP *																																													
REAC-T	X	X			X							X				X		X				X		X		X				X				X	X						X	X		
NORMAREA	X	X																						X																					
NORMVOL	X	X																																											X
Equation Solvers																																													
BPLU	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X
MA18 (35)	X	X				X			X																		X										X								
PGMRES (34)	X	X				X			X																		X										X								
LSOR	X	X																																		X									
Krylov																																											X		
Timestep options																																													
stdy-st	X	X												X			X				X																								
Semi-implicit	X	X	X	X	X	X	X	X		X	X	X	X		X	X		X	X	X		X		X	X	X		X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X
Nearly-implicit	X	X												X			X				X		X																				X		
Hydro-heat explicit	X	X			X			X	X			X	X	X	X				X	X	X	X	X		X	X	X	X		X	X		X	X		X	X	X						
Hydro-heat implicit	X	X	X	X		X	X			X	X					X	X	X											X						X	X					X				
Card 1 options																																													
11 Supercritical	X	X																			X									X														
15 $\Delta t_{Courant}$	X	X			X																										X												X	
23 Godunov	X	X				X																																						
27 MULTID testing	X	X																															X	X										
41K-loss energy dissipation	X	X			X																										X											X	
50 No flip flop	X	X				X																													X							X		
51 No water packing	X	X																								X																		
54 Void truncation	X	X			X																										X											X	
55 Annular mist	X	X			X																										X											X	
Appendix K																																													
Decay heat																							X																					
Metal water reaction																							X																					
Critical flow																							X																					
CHF																							X																					
Post-CHF heat transfer																							X																					

It is recognized that the verification test suite and Features-Cases Matrix are living documents. Both will be expanded as new features and good test cases become available. In particular, the drift.i input deck and several features are not yet included but are left as placeholders.

3.3 Test Cases

Test problems that exercise the important code features and models were identified or created during this task. The test cases were based on current installation problems, problems from the DA[3-1], user problems, or were specifically developed during this task. Many of the installation and DA problems were modified to provide expanded testing of code features. A brief description of each input file is provided in Table 3.1.1. The table identifies the original source of the input file and provides a reference when available. The number of cases that each input file runs is also given.

Table 3.2.1. Input file descriptions.

Input file	Description	Source[1]/ Reference	Cases
2phspump.i	Tests two-phase pump head degradation as a function of void fraction alone and as a function of void fraction and pressure.	I / 3-8	2
3dflow.i	Simulates 3-D flow of single-phase liquid, single-phase vapor, or two-phase flow in a 3x3x3 Cartesian grid with either 1-D or 3-D momentum equations.	I / NA	18
ans.i	Tests decay heat options with the point kinetics model. The problem tests fission power types, fission product types available with each ANS standard, and the G-factor contribution to the decay heat.	I, DA / 3-1	9
boronm.i	Tracks a square wave in boron concentration through a constant area pipe with and without Godunov numerics.	I / 3-2	4
crit.i	Tests Ransom-Trapp and Henry-Fauske critical flow models for a range of stagnation conditions including subcooled, two-phase, and superheated in a small horizontal pipe. Also tests cases with no choking allowed and homogeneous flow.	N / NA	4
cyl3.i	Tests the metal water reaction model for steam flowing past the right surface of a cylindrical heat structure.	I / 3-3	1
drift.i	Not Available yet. Tests the drift flux correlations used to calculate interphase friction for the bubbly and slug flow regimes in vertical components. Tested are all the geometries from Table 6.1-1 of Volume 4. This includes small/intermediate/large pipes, narrow rectangular channels, and bundles. Also tested are all flow ranges, including high and medium upflow, low flow, and medium, high, and very high downflow.	N	0
duklerm.i	Tests the CCFL model using Dukler-Smith air-water countercurrent flow data. Wallis, Kutateladze, and Bankoff correlations are tested.	DA / 3-1	5
eccmix.i	Models a portion of the cold leg of a typical PWR during ECC injection.	N / 3-9	1
edhtrkm.i	Based on the Edward's pipe installation deck, edhtrk.i that simulates a rapid blowdown of a pipe with extras including reactor kinetics, heat structure cosine temperature problems, and many control variables. Deck edhtrkm.i adds all remaining control variable types, except the shaft, and has separate cases to simulate the blowdown of h2o, d2o, h2on, h2o95, hen, and an air/water mixture.	I / NA	5
eflag.i	Simulates blowdown of one vessel into another to check the effect of the e-flag on the thermodynamic state in the downstream vessel.	N / NA	2
enclss.i	Steady-state calculation of a graphite stack using the heat conduction enclosure model.	I / NA	1

fric.i	Tests various single-phase wall and junction friction models. Cases include turbulent flow with and without heated wall effect, laminar flow with and without shape factors, user input equations for wall and form friction, and abrupt area change options.	N / NA	14
fwhtr.i	Represents a tube-in-shell feedwater heater.	I / 3-4	1
gota27.i	Simulates rod-to-rod radiation in a 64-rod bundle in low-pressure steam using the radiation enclosure model.	I / 3-10	1
hse.i	Simulates two-phase flow through a horizontal tee with offtakes coming off the top, bottom, or side face of the horizontal pipe.	N^2 / NA	3
httable.i	Exercises structure boundary conditions related to heat flux and heat transfer coefficient in a simple model of a pipe and heat structure.	N / NA	3
httest.i	Simple model of a pipe and heat structure that achieves various heat transfer regimes for heat transfer packages 1, 111, and 134 by varying initial and boundary conditions. Also tests the non-equilibrium volume option.	N / NA	9
hxco2m.i	Tests the normal and alternate heat structure-fluid coupling models in a steady-state model of a countercurrent, once-through heat exchanger with lead-bismuth on the shell side and supercritical carbon dioxide inside the tubes.	I / 3-12	2
jetjun.i	Simulates insurges and outsurges of liquid into a pressurizer with and without the jet junction model.	N / NA	2
jetpmp.i	Tests jet pump performance over a range of suction and driveline flows.	I / NA	1
l31acc.i	Represents the accumulator response during a slow depressurization during LOFT Experiment L3-1.	DA / 3-1	1
l2-5-emA.i	Tests Appendix K options during a LOFT Experiment L2-5, which simulates a loss-of-coolant accident initiated by a large break.	N / 3-14	1
neptunus20m.i	Models pressurizer insurge/outsurge experiment with spray.	I, DA/3-1	2
pack.i	Vertical fill problem that illustrates the performance of the water packing model when subcooled liquid is injected into superheated steam from below. Tests both the semi- and nearly-implicit options.	N^3 / 3-5	4
pitch.i	Tests an inertial check valve with movement.	I	1
radial.i	Models pure radial, symmetric flow problem in a two-dimensional hollow cylinder. There is no azimuthal flow.	DA / 3-1	1
rcpr.i	Tests the performance of a recompressing compressor in a supercritical CO2 cycle.	I / 3-6	1
refbun.i	Tests two-phase flow and heat transfer with horizontal and vertical bundles that exercise the Groeneveld and PG CHF correlations and correlations for narrow, rectangular channels.	I / NA	1
regime.i	Tests the standard horizontal and vertical flow regimes by adjusting flow boundary conditions through a simple pipe. Both the pre-CHF and post-CHF regimes are tested for the vertical pipe.	N	22
rigidbody.i	Models pure azimuthal, symmetric flow problem in a two-dimensional hollow cylinder. There is no radial flow.	DA / 3-1	1
rtheta.i	Models flow in a two-dimensional hollow cylinder with symmetric flow in both the radial and azimuthal flow directions.	DA / 3-1	1
rtsampnm.i	Tests the axial heat source options using nodal kinetics. Based on typpwr.i, the deck also tests the radio-nuclide transport model.	I / NA	1
rtsamppm.i	Tests various axial heat source options, including those from tables,	I / NA	1

	control variables, and reactor kinetics. This problem is based on typpwr.i and uses point kinetics. The problem also tests the radio-nuclide transport model.		
slab3.i	Tests the metal water reaction model for steam flowing past the right surface of a rectangular heat structure.	I / 3-3	1
sphere3.i	Tests the metal water reaction model for steam flowing past the right surface of a spherical heat structure.	I / 3-3	1
state.i	Tests various fluid states, including subcooled liquid, superheated vapor, two-phase, high-pressure liquid, high-temperature vapor, and supercritical, for h2o, h2on, d2o, and new helium.		24
todcnd.i	Models heat transfer from hot wall with the reflood and two-dimensional heat conduction models.	I / NA	1
turbine9.i	Multi-stage steam turbine with moisture separation. All four types of turbines are tested.	I / 3-7	1
typ1200.i	Models small-break LOCA in a typical pressurized water reactor. Runs 1200 s and tests a variety of models. Based on the original typpwr.i model, but more consistent with current user guidelines.	I / NA	1
typ_kindt.i	TYPPWR input model with nodal kinetics, Krylov solver, and independent kinetics timestep.	N / 3-15	2
valve.i	Models opening and closing of each type of valve, except relief.	N / NA	5
varvol2.i	Uses the variable volume model and a general table to vary the fluid volume of a single-volume filled with liquid.	I / 3-11	1

Footnotes:

1. I = installation problem, DA = DA case from Reference 3-1, N = New input model developed for this task.
2. Based on UP 10044.
3. The water packing model does not come on with the nearly implicit option in Version 403t, which differs from the results presented in Ref. 3-5.

All input decks listed in Table 3.2.1 are used for null testing.

3.4 Verification Directory and Makefiles

The Test suite is organized into a verification directory, Verify, with subdirectories of tests. The running of the tests is controlled by Makefiles.

The main Verify directory contains a single subdirectory for each test case listed at the top of the Features-Decks Matrix. It also contains the principle Makefile, its include files, the template Makefile, set_Makefile, for each of the subdirectories, directories for utility scripts. Within the subdirectories reside the input files for Category 1 and 2 testing, namely the original input deck and a restart deck. There is no Category 3 input file; Category 3 decks are generated from Category 1 decks as described in Section 5. Initially, the only other files that may be present are APT Plot script files.

Makefiles provide several advantages over scripts:

- Gmake allows parallel threading, so two jobs can run on a single core and many jobs can be run simultaneously.
- Dependency lists can be employed to naturally force base case runs whenever the restart file is not available.
- An include-file can set the variables for the Makefile relative to any version of the code and all relevant paths. Then the Makefile remains the same for every code test (new version or updated), only the include file changes for each such test.

The principle Makefile has many functions implemented as Makefile targets. Its major functions are to run null, restart, and backup test suites, and to perform Test (1.1.5) for each suite. The test suite is comprised of the collection of test problems listed in the include file Make.tests. The Make.dirs include file specifies the paths to the Verify-directory, executables, fluid property files, license file, etc.

When the principle Makefile is invoked for null testing from a central platform, such as a machine in the INL cluster, it makes verification runs the following happens:

1. A folder, with the path and name specified by MACHNAME in the include file, is created unless it already exists.
2. Unless one already exists, it creates a folder/directory named Verify within MACHNAME.
3. If a Verify folder already exists in the machine-name-folder, a copy of the folder corresponding to the requested machine is made therein. It has the same name, but with "_old" postpended.
4. Only 2 copies of verification folders of the same name are kept. If there is already an "_old" folder, it will be deleted prior to #3 above.
5. The Makefile moves verification files that are generated to MACHNAME the new Verify folder.
6. It compares matching verification files in two verification folders to produce a single result (success or fail). Comparisons are made via Linux "diff" utility applied after removing date, time, and version information.

The Makefile also has targets for comparing restart and backup runs as explained in Sections 4 and 5. It produces files in the MACHNAME directory named NOTREST and NOTBACK which list the names of the input tests that failed each kind of testing. The Makefile allows the Category 1, 2, and 3 tests to be run separately or all three at once. When testing succeeds, the Makefile gives the following messages:

- For null testing: 'verified'.
- For restart testing: 'Successful Restart Tests'.
- For backup testing: 'Successful Backup Tests'.

3.5 Comparison on Identical Machines

To demonstrate null testing, verification files from RELAP5-3D/Version 4.1.3 run on two different but "identically" configured platforms, FBUILD and FBUILD2 are shown in Figures 3.5.1 and 3.5.2.

Figure 3.5.1. Verification File for Edward's Pipe from INL Enclave Computer FBUILD

```
RELAP5-3D/Ver:4.1.3   fbuild
Time compiled: Aug 14 2013 16:12:07
Date and Time of run: 13/08/14     16:54:20

Case  1  edward's pipe problem base case with extras

Dump     1    Advancement=     109 time=  1.0000E-01
P=      4.9365983737086219E+07 401878A1EFDE58D75B00000000000000
Uf=     1.9649507480408072E+07 40172BD3E37AFC05FEC0000000000000
Ug=     5.4520489485535964E+07 40189FF554BE260AE000000000000000
VOIDg=  7.0158488970410998E+00 4001C103AB179E074A00000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.0448213290728118E+02 400698F6DA1FDA3236D4000000000000
Vg=     2.3165076689908255E+02 4006CF4D3151A9C1FEC1000000000000
RHSth=  0.0000000000000000E+00                                 0
SOLth=  5.2542461771631456E+04 400E9A7CEC6D54CEA4E0000000000000
Error= -8.5282658356481664E-05 BFF165B38EA0ADAA2000000000000000
Temp=   1.1047897158084513E+05 400FAF8EF8B985B33F57500000000000
Flux=   6.4046362410846550E+10 4022DD2EBDE55B16F000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -3.9020138535691576E+00 C000F37530A0CF29DB80000000000000
Cntrl=  3.7065329809843512E+06 4014C47527D90E52D0F595356B020000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

Dump     2    Advancement=     509 time=  5.0000E-01
P=      1.1610017826711973E+07 4016624F43A746CAAC00000000000000
Uf=     1.3706563288757732E+07 4016A24A8693D80DB180000000000000
Ug=     5.3792556235069888E+07 40189A67961E16C52400000000000000
VOIDg=  2.0127747744316551E+01 4003420B4137FFA34180000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.8891214895206032E+02 400720E98297FE2E04B8000000000000
Vg=     9.1675057057565303E+02 4008CA6012B255E284C0000000000000
RHSth=  4.2453960924539154E+07 401843E5E476574C8C12980000000000
SOLth=  1.6144078316381101E+05 40103B50643EB635D838000000000000
Error= -9.9606881069212402E-05 BFF1A1C812FC4B5E8000000000000000
Temp=   1.0939814425864978E+05 400FAB5624EE2286FA5FD00000000000
Flux=   2.7820142401306227E+07 4017A8806E66BC0140000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -1.6980010000000000E+00 BFFFB2B0318B9346980000000000000000
Cntrl=  8.6399604127190748E+05 4012A5DF815219769C2F2BB3AB200000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

CPU Time=  3.1995100000000004E-01 size    2764
```

The comparison shows that only the header and footer information differ. These differences correspond to different computer names, compile times, and execution times in the header and different run times in the footer. When making comparisons for identical calculations, all lines with the keywords *RELAP5* and *Time* should be removed first. Then no differences are registered.

Figure 3.5.2. Verification File for Edward's Pipe from INL Enclave Computer FBUILD2

```
RELAP5-3D/Ver:4.1.3   fbuild2
Time compiled: Aug 14 2013 16:49:37
Date and Time of run: 13/08/14     16:57:21

Case  1  edward's pipe problem base case with extras

Dump     1    Advancement=     109 time=  1.0000E-01
P=      4.9365983737086219E+07 401878A1EFDE58D75B00000000000000
Uf=     1.9649507480408072E+07 40172BD3E37AFC05FEC0000000000000
Ug=     5.4520489485535964E+07 40189FF554BE260AE000000000000000
VOIDg=  7.0158488970410998E+00 4001C103AB179E074A00000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.0448213290728118E+02 400698F6DA1FDA3236D4000000000000
Vg=     2.3165076689908255E+02 4006CF4D3151A9C1FEC1000000000000
RHSth=  0.0000000000000000E+00                                 0
SOLth=  5.2542461771631456E+04 400E9A7CEC6D54CEA4E0000000000000
Error= -8.5282658356481664E-05 BFF165B38EA0ADAA2000000000000000
Temp=   1.1047897158084513E+05 400FAF8EF8B985B33F57500000000000
Flux=   6.4046362410846550E+10 4022DD2EBDE55B16F000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -3.9020138535691576E+00 C000F37530A0CF29DB80000000000000
Cntrl=  3.7065329809843512E+06 4014C47527D90E52D0F595356B020000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

Dump     2    Advancement=     509 time=  5.0000E-01
P=      1.1610017826711973E+07 4016624F43A746CAAC00000000000000
Uf=     1.3706563288757732E+07 4016A24A8693D80DB180000000000000
Ug=     5.3792556235069888E+07 40189A67961E16C52400000000000000
VOIDg=  2.0127747744316551E+01 4003420B4137FFA34180000000000000
QUALa=  0.0000000000000000E+00                                 0
Boron=  0.0000000000000000E+00                                 0
Vf=     2.8891214895206032E+02 400720E98297FE2E04B8000000000000
Vg=     9.1675057057565303E+02 4008CA6012B255E284C0000000000000
RHSth=  4.2453960924539154E+07 401843E5E476574C8C12980000000000
SOLth=  1.6144078316381101E+05 40103B50643EB635D838000000000000
Error= -9.9606881069212402E-05 BFF1A1C812FC4B5E8000000000000000
Temp=   1.0939814425864978E+05 400FAB5624EE2286FA5FD00000000000
Flux=   2.7820142401306227E+07 4017A8806E66BC014000000000000000
dtsum=  3.0000000000000001E-03 3FF689374BC6A7EFA000000000000000
Trips= -1.6980010000000000E+00 BFFFB2B0318B9346980000000000000000
Cntrl=  8.6399604127190748E+05 4012A5DF815219769C2F2BB3AB200000
Rdc:Crnt,Extrp,Mass,Prop,Qual=     0     2     0     2     0
Rpt:Air,DelP,Flip,Jpack,Vpack=     0     0     0     0     0

CPU Time=  3.3494900000000000E-01 size    2764
```

3.6 References

3-1. P. D. Bayless, Editor, "RELAP5-3D Code Manual Volume 3: Developmental Assessment," INEXT-98-00834, Vol. III, Revision 4.0, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, September 2012.

3-2. H. H. Kuo, "Boron Transport," SDVD NRCL2537-29, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, February 1993.

3-3. N. A. Anderson, Metal Water Reaction Model Improvements, SDIVD, INL/MIS-12-27524, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, July 2012.

3-4. G. W. Johnsen and R. A. Riemke, "Verification Test Report Feedwater Heater Model," R5/3D-04-01, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, March 8, 2004.

3-5. R. A. Riemke, "Water Packer Nearly-Implicit Scheme," SDIVD Report, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June 30, 2010.

3-6. C. B. Davis et al., "Modeling the GFR with RELAP5-3D," 2005 RELAP5 International Users Seminar, Jackson Hole, WY, http://www.inl.gov/relap5/rius/presentations.htm, September 7-9, 2005.

3-7. W. W. Weaver, "Upgrade Turbine Model," Verification Test Report, R5/3D-03-05, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June 25, 2003.

3-8. A. D. Hetro, "Two-Phase Pump Degradation Model," SDIVD R5/3D-06-03, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, October 12, 2006.

3-9. S. Z. Rouhani, "ECC Mixer Component in RELAP5/MOD3," EGG-EAST-8813, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, December 1989.

3-10. D. Caraher and R. Shumway, "Enhanced RELAP5/MOD3 Surface-to-Surface Radiation Model," EGG-EAST-8442, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, February 1989.

3-11. W. L. Weaver, "Variable Volume Model, Verification Test Report," R5/3D-98-19, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, November 30, 1998.

3-12. W. L. Weaver, "Alternate Heat Structure – Fluid Coupling Model," Verification Test Report, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June 2006.

3-13. C. B. Davis, "Improved Accuracy for Two-Phase Downflow Scenarios," Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, September 2012.

3-14. C. B. Davis, "Additional Working Fluids in the Appendix K Version of RELAP5-3D," Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, September 2012.

3-15. D. Barber, "RELAP5-3D New Development, Task 1: Asynchronous Time Advancement Methodology," ISL Document No: ISL-ESD-TR-12-03, Information Systems Laboratories, Inc., 2235 East 25th St. Suite 260, Idaho Falls, ID 83415, February 2012.

3-16. C. Gross, "Restart and Backup Testing," Purchase Order Number 7001655, Amendment 3, BMPC, Sep 26, 2012.

4. RESTART TESTING

The RELAP5-3D restart capability[4-1] provides a means to continue a previous calculation from some point after input processing concluded, possibly from the end of the previous run, or an intermediate point in the calculation. Each restart record contains virtually all calculation parameters (E.G. pressures, temperatures, void fractions, flow rates, etc.) for the entire transient calculation[4-2].

The purpose of the restart test suite is to ensure that the restart capability "performed perfectly" for the tested features. By this it is meant that for all problems in the restart test suite, a restart run from an intermediate record in the restart file produces the same verification file dumps as the base case for all common timesteps.

This sort of restart testing has recently been systematized at another national laboratory for another TH code[4-3]. The restart test developed at INL is very similar to that one, including the method of reporting via the Tests-Features Matrix Restart column explained in Section 3.1.

Section 4.1 explains what RELAP5-3D restart is, how it is performed, and what can be done with it. Section 4.2 gives the functional requirements. Section 4.3 explains restart testing with particular attention to the naming of files, particularly when many input cases are in the same deck. Section 4.4 summarizes the high priority user problems that were generated by this task. Section 4.5 is references.

4.1 Background

The restart option is a means to continue a previous RELAP5-3D calculation[4-4] from some point after input processing concluded. This point can be the end of the previous run, the first restart record written immediately after input processing, or some point in between. The previous calculation must be of either NEW or RESTART type[4-1]. The restarted run may be an extension of the previous run or may change the input model before beginning the calculations[4-3]. For an extension restart, the input stream need only contain the control cards that effect the continuation of the problem. For modifications to the input model on restart, new or replacement input cards can produce virtually any necessary model change.

Since restart testing requires the model to remain exactly the same, this latter capability is not allowed in the test suite. The primary control cards present in a Category 2 restart input deck are 100[4-1], 103[4-5], 104[4-5], 199[4-6] and 200-series cards[4-7].

Restart is an important feature to code users and can be used in several ways. When it was developed, computer time and disk storage cost money. It saved money to restart from a saved position so that in the event of failure (code failure, computer loss of power, or exceeding the batch time limit on the run), the calculations could resume from the last restart record. Thus, code calculations were saved to a restart file at multiple times in the transient. There were even built-in controls in the 105-card[4-5] for writing a restart dump a certain number of seconds before the end of the requested CPU-time.

Restarts are employed by code users to perform long-running calculations in pieces[4-4]. A user runs a job overnight, checks the results in the morning, and decides if calculations are proceeding as expected or if some action is needed such as starting over with input changes or modifying of trips, controls, tables, etc. The calculation can be continued in this way with periodic stops and restarts for many cycles.

A common use of restart is to run an input deck to steady-state conditions, then restart the run from that point for a new transient with an operational or accident scenario. The restart is seen as starting the new run at time zero and is often referred to as a “time-zero restart.”

Another possible use of restart is to begin a new run from the middle of a previous run at some point in the transient where plots revealed interesting or unusual behavior. RGUI users sometimes restart with smaller timesteps so that the phenomenon can be observed more closely.

Restart has also been used to circumvent code failure. If a code error causes a failure, in many cases the code can continue past the point of the error if a restart is made from a previous restart record with a different timestep.

Unfortunately, users have reported that the code sometimes does not produce the same calculations on restart as it does when the code had run straight through. This negatively impacts code usability for all of the purposes listed above. Correcting this issue is the primary goal of restart testing.

4.2 Functional Requirements

The functional requirements for restart, or Category 2, testing are summarized as follows:

1. A collection of restart input problems that test all "important" features of RELAP5-3D.
   - Develop a collection of input problems that test the most commonly used code features.
   - This test suite will be used to ensure that the tested features can perform a "perfect restart."
   - This test suite may be smaller than the one used in Task 2.
2. A simple method for performing restart testing
   - Provide an easy and automated means to perform the restart testing.
   - This method executes all of the problems in the test suite at least twice.
     - In the first execution, RELAP5-3D writes a restart record at an intermediate point in the solution.
     - The second execution starts from the intermediate restart.
     - The intermediate value cannot be the initial time-zero restart record.
   - Comparison of verification files at the transient end time must establish the success of the restart.
     - The statement of success or failure must be clear and unambiguous

If any problems fail the restart test, a high priority User Problem shall be submitted to the Idaho National Laboratory.

There are similarities to Category 1 testing in that two different runs are tested by comparing verification files and an unmistakable message is given about the comparison. The important difference is that the two runs that are compared must produce the exact same calculations for the restart to be correct while Category 1 testing may produce differences due to code upgrades and corrections.

4.3 Restart Testing and Naming

As indicated in Section 1, restart testing is similar to Category 1 or null testing in that two runs are compared. These two runs are made, not by comparing verification files from two different versions of the code, but rather by comparing a run to the end of a transient with a run made from a restart record to the same end time. The same test, namely Test (1.1.5) is applied.

A perfect restart test seeks to recognize if restart is seamless. This means the restart run must produce identically the same calculations for all quantities, down to the final bit, as if the calculation had not restarted. Verification files produced by the base-case and restart must be identical.

According to Theorem 1.2.2, restart Test (1.1.5) commits no Type I Error. It has significance level, $\alpha = 0$. This means that the restart will not report differences that do not exist, if the test is programmed correctly.

Consideration should also be given to a second aspect of restart testing, namely version-to-version comparison. Version-to-version restart testing checks if coding updates affect restart performance between two versions by comparing restart verification files between them. The question is whether or not an error between two versions can escape detection by restart testing and necessitate version to version comparison.

Suppose the code produces a perfect restart of incorrect null case calculations. These incorrect results would be found by Category 1 or null testing and would not be found by restart testing by itself since the restart is perfect. This shows that restart testing is not sufficient by itself and must be coupled with Category 1 testing. Alternatively, if the Category 1 test shows null case comparison is perfect, and the restart cases are different, one of the two restarts is not a perfect restart. This will be found by the restart test of one code version or the other when the verification files of the null and restart runs are compared. Again, it is unnecessary to compare restart verification files between two code versions if the null case runs are compared.

All input decks in the verification test suite are candidates for restart testing. Restart files are therefore provided for all the verification test cases listed in Section 3. Input files have specific requirements to test for perfect restarts.

First, timecards of the restart-suite input files must create an intermediate restart dump. It is preferable to select this time so that something interesting is occurring in whatever key parameter the deck tests.

Second, any verification dumps other than the automatic one at the end of the entire transient must occur immediately after the timestep of that intermediate restart. If any dump occurs before the restart, the comparison will show differences because the restart verification file will not have a dump at that time.

Third, for decks that have several input cases, restart files must be named so that each case can be restarted separately. The restart input deck can restart all of the cases.

Other simple differences can be noted. The restart deck must specify restart on the 100 card. It must not contain any cards that initialize anything, and for each case, the title cards must match.

The following naming convention was established. If an input file has a base filename with extension “i” for “input,” then the restart file has the same base filename with “r.i” as its extension. Restart files are named with the case number appended to the base filename after an underscore and use the “r” extension. Plot files have the same name as restart files but use the “plt” extension. This is summarized in Table 4.1.1 for both a single case deck and for decks with other input cases.

For the single case deck, the same restart file, Base.r, is used for the original run and the restart. For a multiple case deck, there is one restart file for each input case. It is named with its case number appended to the base name after an underscore. For each restart file there is a corresponding plot file named like the restart file but with the “plt” extension. Unlike the numerous restart and plot files generated for input

cases, there is only one verification file with all dumps for all cases contained within. It has the "vrf" extension.

Table 4.1.1. Restart Suite Naming Convention

Single Case Deck					
Original input run	Base.i	Base.p	Base.r	Base.plt	Base.vrf
Restart input run	Base.r.i	Base.r.p	Base.r	Base.r.plt	Base.r.vrf
Multi Case Deck					
Restart of 1st case	Base.r.i	Base.r.p	Base_1.r	Base_1.plt	Base.r.vrf
Restart of 2nd case			Base_2.r	Base_2.plt	
Restart of nth case			Base_n.r	Base_n.plt	

The restart input is stored in the same directory as the input file for the null test. This facilitates the comparison of the verification files which can be stored in the same places.

The restart and original deck verification files of a given input model, Base.vrf and Base.r.vrf , are compared to detect imperfect restarts. The comparison is similar to the version-to-version comparison of Section 3 in that all the same lines of the verification file, such as those recording time, are excluded. The test fails for a given input model if there are differences between the two files.

If differences are noted, the name of the input file is recorded in the main verification directory in a file named NOTREST. After all test cases are run and comparisons are made, NOTREST is examined. If it is empty, the restart testing was considered successful and the success message is given. Otherwise a failure message for the restart testing is given.

Implementation of the testing is carried out through Makefiles. As with null testing, the principle Makefile is located in the main verification directory, Verify, and performs the verification testing in the same way. It accesses two include files in Verify, named Make.tests and Make.dirs. The latter lists the location of the executables, fluid properties files, and other auxiliary files. The former lists the restart test directories/tests. **All the problems used in null testing are restarted.** Thus all code features listed in Section 3.1 are restart tested.

The user can, of course, delete names from the list or add directories with a null case and restart deck. The addition or deletion of files from the restart test suite can be accomplished by changing the list in Make.tests. The list can also be overridden on the "make" command line with whatever target the user selects.

When the principle Makefile runs, it links a copy of the test case Makefile, called set_Makefile and located in Verify also, to each test directory. The test directories typically contain the input files for the null test and the restart test though other files are sometimes included, such as APT Plot scripts.

The set_Makefile can prepare for, make RELAP5-3D runs, and perform a variety of cleanups. It can compare runs and store differences. If there are differences it can report the name of the base case in file NOTREST. The default setting is to do all of this for a restart test.

4.4 Restart User Problems Summary

The restart verification suite has been used to test the restart capability of RELAP5-3D/Version 4.1.2. Many restart problems failed to produce the exact same results. All of these have been turned in as high priority (level 2) user problems. Most have been corrected at the time this report is being written. The list is given in Table 4.3.1.

Table 4.3.1. Summary of High Priority Restart User Problems

UP#	Title of User Problem	Resolution	Version	Priority	Modified
13085	4.1.2 httest.i restart fails in Case 9		4.1.2	2	06/21/13
13084	4.1.2 httable.i restart fails in Case 2	fixed	4.1.2	2	07/10/13
13083	4.1.2 fric.i restart fails in Case 13		4.1.2	2	06/21/13
13082	4.1.2t eflag.i restart deck SOMETIMES fails on Case 2	fixed	4.1.2	2	06/24/13
13054	4.1.2t valve.i restart and base runs seriously differ		4.1.2	2	06/24/13
13053	4.1.2 two phase pump base case verification file differs slightly from restart		4.1.2	3	06/21/13
13052	4.1.2 turbine9.i restart quits with input error message	fixed	4.1.2	2	06/21/13
13051	4.1.2 rtsamppm restart core dumps	fixed	4.1.2	2	06/20/13
13049	Restarts using MA18 and PGMRES do not work		4.1.2	2	06/20/13
13048	4.1.2 jetjun restart and base runs differ on both cases		4.1.1	2	06/20/13
13047	4.1.2 hxco2m restart base and restart differ on Case 2	fixed	4.1.1	2	06/20/13
13046	4.1.2 floreg: verification files of most restart cases differ from base run		4.1.1	2	06/18/13
13045	4.1.2 edhtrkm restart core dump on Case 2	fixed	4.1.1	2	06/18/13
13044	4.2.1t cyl3.i restart slightly differs from base case	fixed	4.1.1	2	06/18/13
13043	4.1.2 crit.i base case verification file differs from restart		4.1.1	2	06/18/13
13042	Boronm restart differs from base case	fixed	4.1.1	2	06/18/13
13041	Restart failure: state.r.i case 11 w/ glibc mem corruption	fixed	4.1.1	3	06/10/13
13040	Restart of httest case 6 fails with memory corruption	fixed	4.1.1	3	06/10/13
13039	Floreg case 8 restart creates glibc memory corruption	fixed	4.1.1	3	07/10/13
13038	Restart core dump for typkindt in idetector		4.1.1	3	06/20/13
13037	Error in restart of l2-5 deck		4.1.1	1	06/05/13
13035	Restart of ans.i0, ans.r.i, takes more advancements.		4.1.1	3	04/26/13
13033	Restart fails for rtsamppm.r.i.	fixed	4.1.1	3	04/29/13
13032	Restart fails for rtsampnm.r.i	fixed	4.1.1	3	04/29/13
13031	Restart of pitch produces unreliable information.		4.1.1	3	04/25/13

Many of those that remain are caused by an inaccuracy in the last bit of the timestep on restart.

4.5 References

4-1 The RELAP5-3D Code Development Team, RELAP5-3D Code Manual Volume V: User's Guidelines, INEEL-EXT-98-00834, Revision 4.0, Section 4.1.2, pp. 4-2, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June, 2012.

4-2 The RELAP5-3D Code Development Team, RELAP5-3D Code Manual Volume V: User's Guidelines, INEEL-EXT-98-00834, Revision 4.0, Section 3.1.4.2, pp. 3-17 to 3-18, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June, 2012.

4-3 D. L. Aumiller, G. W. Swartele, J. W. Lane, F. X. Buschman and M. J. Meholic, "Development of Verification Testing Capabilities for Safety Codes," Section 7, The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, NURETH15-145, Pisa, Italy, May 12-17, 2013.

4-4 The RELAP5-3D Code Development Team, RELAP5-3D Code Manual Volume II: User's Guide and Input Requirements, INEEL-EXT-98-00834, Revision 4.0, Section 8.7, pp. 8-31 to 8-34, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June, 2012.

4-5 The RELAP5-3D Code Development Team, RELAP5-3D Code Manual Volume V: User's Guidelines, INEEL-EXT-98-00834, Revision 4.0, Sections 4.1.5-4.1.6, pp. 4-4 to 4-5, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415 June, 2012.

4-6 G. L. Mesina, "Implementation of a New DTSTEP Algorithm for use in RELAP5-3D and PVMEXEC Completion Report," INL/EXT-11-20798, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, Apr, 2011.

4-7 The RELAP5-3D Code Development Team, RELAP5-3D Code Manual Volume V: User's Guidelines, INEEL-EXT-98-00834, Revision 4.0, Section 4.2, pp. 4-8 to 4-9, Idaho National Laboratory, PO Box 1625, Idaho Falls, Idaho 83415, June, 2012.

5. Backup Testing

A backup is the action of repeating an advancement. The code can back up and use the same time step or a smaller one. An advancement with the same step is often called a timestep repeat and the other is called a timestep reduction. Backup testing seeks to determine if the code can repeat a timestep correctly. This requires resetting conditions to the beginning of timestep values and redoing the advancement. If this is programmed correctly, and the same timestep is used, the transient calculation with and without the backup will produce the same result. If the timestep were cut, the calculation would be different from that point forward. For this reason, only backups *with the same timestep*, namely timestep repeats, are considered. The backup testing performed here is really timestep repeat testing, but shall be referred to as backup testing for simplicity.

There are 2 situations that can cause the code to repeat an advancement without timestep reduction:

1. Direction from the PVM Executive.
2. Internal conditions that cause the code to set its "success" flag to 5

PVM Executive direction is not considered in this task. This task is specific to RELAP5-3D alone.

Backup testing seeks to determine if a "perfect" code backup can be made on any given timestep. The user artificially induces a backup by causing RELAP5-3D to set its flags to indicate a backup condition, even though no actual conditions that cause a backup exist. In theory, the code will produce the same calculations on both attempted advancements, the first and the one it is forced to take because of the artificial setting of the backup condition flags.

To examine code backup capability, Test (1.1.5) is applied. Two runs are made, one uses the original deck of a given input model and the second uses a copy of that deck with a forced backup. The two runs are compared and conclusions drawn.

According to Theorem 1.2.3, the backup test will not indicate that there are differences when none exist (if the verification programming is correct). However, it is possible that differences do exist but are not found by the test. The power of Backup Test (1.1.5) can be increased by increasing the thoroughness of the testing. There are three ways to do that:

- Increase the number of variables written to the verification file
- Increase the features of the code tested in the backup test suite
- Increase the number of forced backup runs for a given input deck

Section 5.1 gives some background on code backups and the functional requirements. Section 5.2 explains some of the intricacies of testing backups. Section 5.3 describes how backup testing is implemented in the code. In Section 5.4 special considerations for construction of the test input decks are discussed. The results of applying backup testing to RELAP5-3D, including the User Problems it generated, are found in Section 5.5. Section 5.6 lists the references.

5.1 Background

Three mechanisms cause the code to back-up and repeat an advancement *with the same timestep*:

1. Appearance of a noncondensable in a control volume (often called air appearance)[5-1].
2. Velocity flip-flop[5-2] at a junction
3. Water packing/stretching[5-3] in a control volume

The code takes the condition into account when building the discrete form of the governing equations[5-1, 5-2, 5-3], producing a system whose solution better reflects the physical conditions that cause backup.

5.1.1 Noncondensable Gas Backup Summary

The appearance of a noncondensable gas in a control volume that had none on the previous timestep causes problems with the derivatives of the phasic properties with respect to X_n, the noncondensable gas quality. At the beginning of the advancement, the derivatives are zero since there is no such gas in the volume. At the end of the advancement, the derivatives would need to be zero for a consistent state as described in the RELAP5-3D Code manuals[5-1], but are not. The situation is detected whenever the noncondensable quality in a volume is zero at the beginning of the advancement and the flux of noncondensable gas into the volume during the advancement is greater than a small noncondensable source term.

When this condition is detected, an explicit mass balance on the gas phase in the volume is used to estimate the noncondensable quality and gas fraction that would result from the flux of noncondensable gas into the volume. The derivatives of the phasic properties are computed from the estimated noncondensable gas quality. The backup uses these and the estimated gas fraction to construct the discretized governing equations when the timestep is repeated.

5.1.2 Velocity Flip-flop Backup Summary

For velocity flip-flop[5-2], the convective terms in the governing equations are calculated with donored properties determined by the direction of the phasic velocities. When the final velocities differ in direction from the explicit velocities used to define the donored properties, mass and energy errors may result due to the incorrect properties used in the discrete system of equations. A velocity flip-flop has occurred when one of the junctions, j, in a system satisfies the following (mass and energy) condition:

$$\left|A_j - B_j\right| > 0.2\ A_j \tag{5.1.1}$$

where

$$A_j = \alpha_{fj}^{n,\exp} \min\left(\rho_{fj}^{n,\exp}, 5\rho_{gj}^{n,\exp}\right) U_{fj}^{n,\exp} + \alpha_{gj}^{n,\exp} \rho_{gj}^{n,\exp} U_{gj}^{n,\exp}$$

$$B_j = \alpha_{fj}^{n+1} \min\left(\rho_{fj}^{n+1}, 5\rho_{gj}^{n+1}\right) U_{fj}^{n+1} + \alpha_{gj}^{n+1} \rho_{gj}^{n+1} U_{gj}^{n+1}$$

The junction liquid fraction, liquid density, liquid specific internal energy, void fraction, gas density, and gas specific internal energy, respectively, $\alpha_{fj}^{n,\exp}$, $\rho_{fj}^{n,\exp}$, $U_{fj}^{n,\exp}$, $\alpha_{gj}^{n,\exp}$, $\rho_{gj}^{n,\exp}$, $U_{gj}^{n,\exp}$ are based on the beginning of timestep explicit velocities, denoted by the superscript "n,exp." The same variables from the end of the calculation have the superscript n+1.

The backup repeats the advancement with the same timestep size but using the donor properties based on the calculated final velocities.

5.1.3 Water Packing Backup Summary

Consider a control volume that is nearly full of liquid with a net influx of mass. The density-pressure relationship used to calculate the new time pressure uses the beginning of timestep values for the state properties and derivatives. The compressibility of this low void fraction control volume is dominated by mass transfer. However, the mixture of liquid and vapor/gas corresponds to a highly compressible fluid; that is, significant volume changes can easily occur with very little change in pressure.

It can happen that on some advancement, the net volume influx of liquid during the timestep can be larger than the vapor/gas volume in the cell. On the following timestep, since liquid influx momentum from the beginning of the timestep is used, and because the liquid is nearly incompressible, the large liquid influx into cell filled with liquid causes a *large* increase in pressure.

The computed pressure spikes caused by water packing are unphysical. In the neighborhood of these spurious numerical pressure spikes, the calculated phasic velocities may increase substantially, reducing the material Courant stability limit, resulting in smaller time-step sizes and reduced computational efficiency. Furthermore, water packing may severely distort the transient solution by changing the void distribution or driving the liquid completely out of an open system.

Water packing occurs in volume K whenever all the following conditions hold:

- the void fraction, $\alpha_g \leq 0.12$
- liquid temperature = $T_f < T_s$ = the saturation temperature
- volume K is vertically stratified
- the volume above it is highly voided
- and $P_K^{n+1} \geq P_K^n + 0.0023 P_K^n$

where n is the advancement number at the beginning of the timestep.

The code adjusts the discrete system as presented in RELAP5-3D code manual Volume 1[5-3].

5.1.4 Value of Code Backup

Backup is an important code capability because it saves run time and prevents failures. Tracing the logic flow in subroutine TRAN provides understanding of how the algorithm works. Currently, when the code detects a backup condition, it jumps to the end of the transient loop (thereby bypassing all remaining transient calculations including kinetics, PVM transfers, and control system calculations), restores the data to beginning-of-timestep values (in DTSTEP), and returns to the top of the transient loop to starts the timestep over with certain flags set to indicate which kind of backup to perform in order to adjust the discrete system properly. This ensures that the restoration of data is correct. Moreover, the code saves CPU time. However, backing up provides even greater savings.

As discussed in Section 5.1.3, if backups were not a code capability, the only other option would be to halve the timestep, restore the data, and begin the timestep over. To understand the time-savings of backup over halving the timestep, consider what would happen if the code recognized backup conditions just as it does now, but applied halving instead of backup.

When a *backup condition* occurred, the code would halve the timestep and attempt to advance. The timestep algorithm requires two consecutive successful steps before doubling the timestep size. If the first step succeeded, the code would attempt a second advancement with the same timestep. If the second step succeeded, the code would take twice as much computer time as one successful step with the backup algorithm. However, there is no guarantee the backup condition would disappeared on either of the two attempted advancements merely because the timestep was cut. If the condition was detected again, the code would continue to halve the step size until an attempted advancement succeeded. If the step is cut

twice, at least three advancements (two steps at ¼ the original size and one at ½ the original) are required to advance as far as the original timestep.

Without backup logic, it is possible that the minimum timestep could be reached with the backup condition still present. Unable to cut the timestep further, the code would be forced to print an error message and stop, just as it does for a thermodynamic property error at minimal timestep size.

Even if the backup condition is mitigated by rebuilding the discrete governing equations with an adjustment for the backup condition, one of the other two backup conditions can appear immediately after the first *and* on the same attempted advancement. If that happens, the code takes another backup to clear the conditions. After three backups at the same cumulative time, the code cuts the timestep. This prevents the code from endlessly cycling between different backup conditions.

The code is much more computationally efficient with backup logic than without. Historically, the code once had two forms of backup, partial and full[5-5]. Full backup is described above and was used in PVM coupled calculations and in some nearly-implicit calculations. Partial backup sought to save even more computational time by not returning to the beginning of the timestep, but instead backing up within the hydrodynamic calculation far enough to rebuild the discrete hydrodynamic system for each type of backup. This saved computer time, but had numerous implementation issues. It was eventually abandoned for the simpler full backup algorithm.

5.2 Functional Requirements

The original functional requirements for automated backup testing are summarized as follows:

1. A collection of input problems that test all "important" features of RELAP5-3D
   a. Develop a collection of input problems that test the most commonly used features in RELAP5-3D
      i. Use the test suite to ensure that the tested features can perform a Perfect restart
   b. This test suite can be smaller than the one used in Category 1 testing
   c. All code features that are absent from the Category 1 test suite must be identified.
2. A simple method for performing backup testing
   a. Provide an easy and automated means to perform the backup, Category 3, testing.
   b. This method must execute all of the problems in the test suite at least twice.
      i. In the first execution, RELAP5-3D must fail timesteps at a user specified time or timestep number, and repeat those timesteps at the same timestep size.
      ii. The second execution must execute without the artificial backups.
   c. Comparison of the verification results at the transient end time must establish the success of the backup process.
   d. In selection of the times to select for backup testing, consideration must be given to special processes that may have special backup requirements.
      i. Examples of such processes include: Entry and exit of choked flow; Entry and exit of CCFL; Insertion and removal of fine-mesh heat transfer nodes in a reflood model.
3. The list of test problems provided for this task shall include the times or timestep numbers that are used for backup testing. This method must provide an unambiguous statement concerning the success, or lack of success, of the backup testing.
4. If any problems fail the backup test, a high priority User Problem must be submitted to the Idaho National Laboratory.

A modification of the functional requirement 2d has been made to allow a more thorough method for testing backup logic. The method forces back on every successful timestep. The reasons that this is superior are explained in Section 5.3.

5.3 Conceptual Intricacies of Backup Testing

To test backup, the user compares the results of a normal run with those calculated when input directs the code to perform an artificial backup on one or more timesteps.

A natural backup occurs when the code recognizes that one of the three conditions listed in Section 5.1 has occurred after the discrete system has been solved. It sets a variable named "success" to 5 and a flag named lpd(i)%lsucces to 5 (or possibly some other values that are not relevant here), where i is the number of the hydrodynamic system (primary side of a power plant, secondary side, etc.) in which the condition occurred. New programming allows these same two flags to be set to five via user input to artificially create the backup signal.

Although there are three different mechanisms that cause a backup, in theory they are all equivalent for forcing an artificial backup. In an artificial backup, neither the data nor the calculations that form the discrete system of equations are supposed to differ. The discrete system should therefore be the same for all three artificial conditions. Moreover, it should be the same as the system produced for the previous normal attempted advancement of that same timestep simply because the values contained in the arrays and other variables used to construct the discrete system should be the same.

Backup testing seeks to find if this is indeed the case. Test (1.1.5) compares verification files from two runs, the normal run and the forced backup runs. Verification dumps could be made at specified times but always occur on the final step. If the verification files have no differences, then the code has performed a perfect backup. If not, a failure must be reported by the testing system and a high priority user problem report generated.

It is important that the artificial backup be generated only on steps that have no natural backup. This avoids causing a timestep cut from too many consecutive backups on the same advancement.

It is preferable to perform a backup on a timestep when the code is experiencing some other stressful condition. Examples of such conditions include: entry and exit of choked flow; entry and exit of CCFL; insertion and removal of fine-mesh heat transfer nodes in a reflood model. Though such conditions will test the code under the most trying of circumstances, it is insufficient. For example, the same input model may succeed when forced to backup at one time but not at another, so testing at a single point in time could miss a code error. Also, developmental changes to the code can affect when the stressful conditions occur. After developmental updates, the same test deck may miss the intended conditions entirely.

A much more thorough test ignores such conditions and forces backup at every timestep that does not have a natural backup. This process was developed at another national laboratory[5-4]. It guarantees that backups are forced on the most stringent conditions that occur, and on every other kind of condition that the model generates. This procedure was chosen as the default form of testing.

It must be noted that such thorough testing causes the test set to run much longer. Test cases with forced backups on all advancements that have no natural backups tend to run almost twice as long as the corresponding original case. And the original case must be run as well. Therefore it is important to have short-running problems in the test set.

The verification files from normal and backup runs can differ in several ways even when the two runs produce exactly the same calculations.

1. The backup lists the user backup type (air appearance, flip-flop, etc.) on a line of output.
2. The line that records the number of repeat conditions should be different.
3. The number of attempted advancements will be different.
4. The number and time of verification dumps can be different.

To prevent detection of these nonessential differences, lines associated with items 1-3 are eliminated from the verification files (null and backup) before the files are compared.

Difference number 4 is complex. Consider comparing the verification dumps between an original and forced backup run. A normal run must produce dumps at the same cumulative times as the forced backup run. This can be done using advancement count because a linear relationship between the advancement counts of the two runs exists as long as no natural backups occur. However, if one or more do occur, the correspondence between the numbering of *attempted* advancement between the two runs becomes more complex. Comparison based on attempted advancement count cannot be pre-programmed easily.

An alternative is to employ *successful* advancements rather than *attempted* advancements. The former are not affected by backup conditions. Therefore, the meaning of 199-card Word 3, when it is an integer, is successful advancement rather than attempted advancements as the code interprets it for Category 1 and 2 testing.

Another consideration for difference number 4 above is that if two or more different points in time are selected for backup testing in a single input model, then a pair of runs must be made for each test in order that the normal run have dumps on the *same* timesteps to those produced by the forced backup run, otherwise a simple comparison of the verification files will show differences even if none exist. Moreover, if dumps are made on every timestep of a thorough test that has backups on all timesteps, its verification file could be large or exceed the 1 MB limit.

To overcome this issue associated with difference number 4, it was decided that thorough backup test runs (there is a backup on every timestep, natural or forced) cannot dump to the verification file except on the final timestep which is automatic. Other forms of backup testing perform dumps at their specified times.

This solves many problems in a simple manner and reduces file storage considerably. The initial backup test is run using thorough testing. If differences are detected, one of the other three conditions can be specified (air appearance, water packing, velocity flip-flop) and the code will make dumps at specified times to compare against a normal run (which must make the same dumps). The way to specify these options through code input are explained in the next section.

5.4 Backup Code Implementation

To implement verification backup testing, a new subroutine was written and many were modified. This section describes these developments.

5.4.1 Transient Coding

Tracing the logic flow of a naturally-occurring backup helps understand where to modify the code to implement backup testing.

The transient controlling subroutine, TRAN, processes success conditions immediately after the call to subroutine HYDRO. As indicated in Section 5.1, TRAN ignores all remaining parts of the transient and proceeds directly to the DTSTEP section if the success-flag is 5. Within DTSTEP, first the termination and repeat conditions are processed in Section 2. When the success-flag is 5, the code moves the old time data into the new time locations, backs cumulative time up by one timestep, and proceeds to Section 11, the finalizing stage. There the cumulative time advance by just as much as it was backed up in Section 2, but attempted advancement count increases. Control returns to TRAN. In TRAN, the end of the time loop is encountered and processing begins again at the top with the same cumulative time as in the previous pass through the loop.

A new subroutine VERFBACKUP was written. It ensures that backups occur only:

- when backup testing is requested by the user
- on the user-specified time or times
- after a successful advancement
- when no natural backup occurs on the timestep
- when the verification file size limit is not exceeded

Subroutine VERFBACKUP sets the two success-indicator flags, *succes* and the flag *lpd(1)%lsucces* for hydrodynamic system 1 (system 1 always exists if there are any hydrodynamic control volumes) and the appropriate backup condition flag or flags on backup steps. When it forces backup it writes a message on the printed-output file about the backup. It also remembers the cumulative time of the previous backup and does not force the code to fake a backup until cumulative time increases. It does not interfere with a legitimate backup, but will force one after a successful timestep.

In addition, it sets the following flags:

- lpd(1)%airap = 1 for air appearance
- lpd(1)%lpackr = 1 for water packing
- lpd(1)%vlflip = 1 for velocity flip-flop
- lpd(1)%airap = 1 for thorough testing with backups on all steps

Subroutines HYDRO calls VERFBACKUP immediately after the call to VFINL or VIMPLT when the velocity field has been determined. HYDRO calls it only when verification testing is active.

Subroutine VERFSUM was modified to make no dumps on the verification file (call VERF_DUMP) on non-terminal timesteps when verfaction is 5 (thorough all advancements backup testing).

5.4.2 Backup Input Coding

Subroutine RDEBUG was upgraded to include the new verification options for backup testing. As in previous sections, this is implemented with the 199 card. The 199 card format is:

199 verify condition start end

The condition keyword specifies the mechanism (air appearance; water packing or stretching; and velocity flip-flop) that causes backup. These are the naturally occurring backup conditions as reported in Section 5.1. A condition keyword, backall, is also provided to specify the thorough testing option which performs a backup after every successful advancement. The 199 card "condition" keywords for these testing mechanisms are summarized in Table 5.3.1.

Table 5.3.1. 199 Card Condition Keywords for Backup

Cause	Condition Keyword
air appearance	backair
velocity flip-flop	backvel
water packing	backpck
Thorough testing	backall

Module VERIFYMOD was modified to incorporate the backup testing feature. Variable *verfaction* was expanded to have values to indicate backup conditions according to Table 5.3.2.

Table 5.3.2. Values of variable *verfaction*.

Value	Description
0	No verification
1	Verification dumps requested by user
2	Backup for air-appearance
3	Backup for water packing
4	Backup for velocity flip-flop
5	Thorough testing, backup after every successful advancement

The write statement in VERFBACKUP verifies that the code does backup when *verfaction* has any of the values 2-5.

When *verfaction* is 5, the coding makes backup verification dumps on the final step only. This allows a single base case run to be compared with numerous backup runs at different advancements or times.

5.5 Input Decks for Backup Testing

An important aspect of backup testing is the formation of the input decks. There must be two decks, a backup input deck and an appropriate copy of the null test input deck. *The backup input decks are generated from the null test deck at the time of the testing*. This is the only way to ensure that the null and backup decks match perfectly in all ways (except for the two 199 cards that force backup testing).

It was noted above that "backall" testing makes a single verification dump on the final step only. This arrangement simplifies the coordination of non-backup and backup run output. However, normal input deck of the verification suite is designed to compare against the *restart* run. It produces two verification dumps, the automatic one on the final step and one right after a selected restart write. It has an extra dump that the "backall" run does not have.

Therefore a copy of the normal input deck is made with its 199 card modified to dump on the final step only. A second copy of the normal deck is created with its 199 card condition set to "backall."

The following naming convention was established. If the normal input file is named Base.i, then the corresponding input files for backup testing are named Base.b.i and Base.bk.i. As with restart, the normal file extensions replace the "i" for the other output files. This is summarized in Table 5.4.1.

Table 5.4.1. Backup Suite Naming Convention

Base case input file	Base.b.i	Base.b.p	Base.b.r	Base.b.plt	Base.b.vrf	Base.bak
Backup of base case	Base.bk.i	Base.bk.p	Base.bk .r	Base.bk.plt	Base.bk.vrf	none

The two backup input are generated and stored in the same directory as the *input file for the null test.*

5.6 Backup Testing

The backup and null case verification files of a given input model are compared to detect imperfect backups. The comparison is similar to the version-to-version comparison of Section 3, excluding all the certain lines of the verification file *before* the comparison is made. These lines contain unimportant or non-calculated differences:

- Lines with date and time
- Number of repeats
- The RHS and Solution sums
- The backup information (which is not in the null case run).

The test fails for a given input model if there are differences between the two files. If differences are noted, the name of the input file is recorded in the main verification directory in a file named NOTBACK. After all test cases are run and comparisons are made, NOTBACK is examined. If it is empty, the restart testing was considered successful and the success message is given. Otherwise a failure message for the restart testing is given.

Implementation of the testing is carried out through Makefiles, the same ones for all three categories of testing (null, restart, and backup), namely Makefile and set_Makefile in the main verification directory, Verify. There are two include files, Make.tests that lists all the test directories and Make.dirs that specifies the directories for the fluids, RELAP5 executable, license file, etc.

The input decks backup-tested are listed in Make.tests. The user may modify this list or override it by changing the content of the backup list when running the make command. **The default setting is that all decks in the verification test suite are backup-tested**.

The principle Makefile links a copy of set_Makefile to each test directory with local name Makefile. It then invokes those Makefiles in turn to perform the backup testing. Those Makefiles create two copies of the base case input deck, Base.b.i and Base.bk.i, as listed in Table 5.3.2. Base.bk.i is the original deck, Base.i, with the 199 card modified with the backall keyword. With backall, the only verification dump is on the final timestep. Base.b.i is the original deck, Base.i, with the 199 card modified to dump the final advancement only. Thus the two verification files have the same dumps to compare.

The system has been tested. Initially, nearly all backup problems failed to produce exactly the same results. All of these cases have been turned in as high priority (level 2) user problems. Many have been corrected at the time this report is being written. The summary of the user problems is given in Table 5.5.1.

Table 5.5.1. Summary of High Priority Backup User Problems

UP#	Title of User Problem	Resolution	Version	Priority	Modified
13081	4.1.2t floreg.bk.i Backup deck hangs computer	fixed	4.1.2	2	06/20/13
13080	4.1.2t rcpr.i Backup deck hangs computer	fixed	4.1.2	2	06/20/13
13079	4.1.2t ans.i backup deck fails on Case 6		4.1.2	2	06/20/13
13078	4.1.2t state.i backup deck fails on Case 7		4.1.2	2	06/20/13
13077	4.1.2 valve.i base and backup cases differ		4.1.2	2	06/20/13
13076	4.1.2 turbine9.i base and backup cases differ		4.1.2	2	06/20/13
13075	4.1.2 todcnd.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13074	4.1.2 sphere3.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13073	4.1.2 slab3.i base and backup cases differ	fixed	4.1.2	2	07/08/13

UP#	Title of User Problem	Resolution	Version	Priority	Modified
13072	4.1.2 rtsamppm.i base and backup cases differ		4.1.2	2	06/20/13
13071	4.1.2 reflecht.i base and backup cases differ		4.1.2	2	06/20/13
13070	4.1.2 refbunm.i base and backup cases differ		4.1.2	2	06/20/13
13069	4.1.2 pitch.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13068	4.1.2 pitch.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13067	4.1.2 neptunus20m.i base case differs from backup		4.1.2	2	06/20/13
13066	4.1.2t jetjun.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13065	4.1.2t hxco2m.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13064	4.1.2 httest.i base and backup cases differ		4.1.2	2	06/20/13
13063	4.1.2t hse.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13062	4.1.2t gota27.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13061	4.1.2t fwhtr.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13060	4.1.2 enclss.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13059	4.1.2 eflag.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13058	4.1.2 edhtrkm.i base and backup cases differ	fixed	4.1.2	2	07/08/13
13057	4.1.2t duklerm.i base and backup cases differ		4.1.2	2	06/20/13
13056	4.1.2 cyl3.i base and backup cases differ	fixed	4.1.2	2	07/05/13
13055	4.1.2 backup problems with slight differences	fixed	4.1.2	2	07/05/13
13050	4.1.2 rtsampnm.i base case core dumps	fixed	4.1.1	2	07/10/13
13030	Restart of neptunus20m produces unreliable information.		4.1.1	2	04/25/13
13029	Jet pump restart reads to end of file w/o finding record.		4.1.1	2	04/25/13
13028	Restart two-phase pump, Case 1, not accurate.		4.1.1	2	04/25/13
13027	fwhtr.i1 reads to the end of the restart file w/o finding record	fixed	4.1.1	2	04/25/13
13026	Restart of varvol2 produces slight difference on final verification dump		4.1.1	5	04/25/13
13025	Restart of slab3.i produces some unreliable information.	fixed	4.1.1	2	04/24/13
13024	refbunm takes different no. of advancements for base case & restart	fixed	4.1.1	2	04/24/13
13023	Failure in reflecht.i1	fixed	4.1.1	2	04/24/13
13022	Slight problem with restart of L2-5-emA		4.1.1	2	04/24/13

5.7 References

5-1 The RELAP5-3D Code Development Team, "RELAP5-3D Code Manual Volume I: Code Structure, System Models and Solution Methods," INL-EXT-98-00834-V1, Revision 4.0, Section 8.2, p 8-4, June, 2012.

5-2 The RELAP5-3D Code Development Team, "RELAP5-3D Code Manual Volume I: Code Structure, System Models and Solution Methods," INL-EXT-98-00834-V1, Revision 4.0, Section 8.2, pp. 8-3 to 8-4, June, 2012.

5-3 The RELAP5-3D Code Development Team, "RELAP5-3D Code Manual Volume I: Code Structure, System Models and Solution Methods," INL-EXT-98-00834-V1, Revision 4.0, Section 3.4, pp. 3-271 to 3-274, June, 2012.

5-4 D. L. Aumiller, G. W. Swartele, J. W. Lane, F. X. Buschman and M. J. Meholic, "Development of Verification Testing Capabilities for Safety Codes," Section 8, The 15th International Topical Meeting on Nuclear Reactor Thermal - Hydraulics, NURETH-15, NURETH15-145, Pisa, Italy, May 12-17, 2013.

5-5 R. A. Riemke, "Replace the Partial Backup Logic with Full Backup Logic," Idaho National Laboratory document R5/3D-04-02 Rev. 0, Mar. 10, 2004.

6. SOURCE CODE

This section describes the coding that implements verification testing. It also includes the actual coding of new subroutines. In the case of modifications to existing routines, small explanations of explanation of the changes are given.

The coding was written to isolate the data and calculations from the rest of RELAP5-3D as much as possible. The module uses only two level modules, intrtype and ufilsmod. Thus it can be built at any point after those kinds of modules. Any data needed from any other module comes into its subroutines through call sequences. Any verification-related subroutines that required a significant amount of data from other modules were made separate from VERIFYMOD.

Further, the interface into RELAP5-3D of the new coding was minimized. Thus very little besides a call to a verification subroutine and references to verification scalars is used to access the verification coding from any existing routine. The only exception to this is RDEBUG which was wholly rewritten.

The input coding was written to spot user input errors and give good warning and error messages, as is consistent with the rest of RELAP5-3D input processing.

Section 6.1 presents the source code for verifymod.F90. Section 6.2 reproduces the source code for redbug.F. Section 6.3 lists the source code for verfbackup.F

6.1 Module VERIFYMOD

Each internal subroutine sums the number of bytes it writes on the verification file. VERF_LAST shuts down further writes if the 1 MB limit is exceeded in verfsizlim. Thereafter, only the final write of each remaining input case may be written to the verification file.

```
module verifymod
!
!  Module associated with verification testing
!
!  1.0  Declarations
!
 use intrtype
 use ufilsmod, only: verifl
 implicit none
!
 real (sdk) :: backtime, beginVerf
 integer (sik) :: caseDmpCnt, dumpStep
 integer (sik) :: verfDumpLeft, verfaction, verfspace = 0
 integer (sik), parameter :: verfsizlim = 1048576 - 1351
 logical :: ljunflag(8), lverify, lvolflag(7)
 character (12) :: compileday, compiletime
 character (64) :: compilehost
!
 type verf
   integer (8) :: rdccrnt, rdcextr, rdcmerr, rdcprop, rdcqual
   integer (8) :: rptair, rptdpr, rptflip, rptjpack, rptpack
   real*16 :: thsolnsum, psum, vfsum, vgsum
   real*16 :: boronsum, qualasum, ufsum, ugsum, voidgsum
   real*16 :: tempsum
   real*16 :: fluxsum, minorsum
   real*16 :: rhsthsum, rhshesum, rhsnksum
   real*16 :: dtsum, errsum, cntrlsum, tripsum
 end type verf
!
 type (verf) :: vrf
!
!  2.0  Data Dictionary
!
!  2.1  Scalars
!  backtime   = cumulative TIME of previous BACKup
!  beginVerf  = BEGINning VERiFy cumulative time on which to activate
!               -1.0  if  integer-trigger used & active-to-transient-end
!  caseDmpCnt = COUNT of the number of VERiFication dumps
!  compileday = DAY on which the code was COMPILEd
!  compilehost= HOSTname of machine on which the code was COMPILEd
!  compiletime= TIME at which the code was COMPILEd
!  dumpStep   = step on which previous verification dump was written
!  ljunflag   = Logical JUNction FLAG (jefvcahs)
!               For Verification Table Property Rows, indicate if a model
!               is turned in in the input model during input edit.
!  lverify    = Logical VERIFY flag
!               true - on this timestep, do verify calcs and a dump
!               false - do NOT perform verification calculations
!  lvolflag   = Logical VOLume FLAG (tlpvbfe)
!               For Verification Table Property Rows, indicate if a model
!               is turned in in the input model during input edit.
!  verfaction = ACTION for VERiFication file: user-requested/condition-based
!               -1 - Close Verifl
!               0  - Error: set fail flag
!               1  - Open/dump Verifl
```

```
!               2  - Open/backup due to air appearance
!               3  - Open/backup due to water packing
!               4  - Open/backup due to velocity flip-flop
!  verfSpace  = amount of SPACE in bytes left on the verify file
!
!  2.2  Derived Type
!
!  TYPE verf
!
!  INTEGER DATA
!  rdccrnt  = number of times the time step was ReDuCed by the CouRaNT
!             limit in all volumes since the beginning of the transient.
!  rdcextr  = number of times the time step was ReDuCed by the state
!             EXTRapolation in all volumes since the beginning of the transient.
!  rdcmerr  = number of times the time step was ReDuCed by Mass ERRor
!             in all volumes since the beginning of the transient.
!  rdcprop  = number of times the time step was ReDuCed by water PROPerty
!             error in all volumes since the beginning of the transient.
!  rdcqual  = number of times the time step was ReDuCed by QUALity adjustment
!             in all volumes since the beginning of the transient.
!  rptair   = total number of time steps RePeaTs due to AIR appearance
!  rptdpr   = total number of time steps RePeaTs due to PRessure change (Delta)
!  rptflip  = total number of time steps RePeaTs due to velocity FLIP-flop
!  rptjpack = total number of time steps RePeaTs due to Junction water PACKing
!  rptpack  = total number of time steps RePeaTs due to volume water PACKing
!
!  REAL DATA
!  boronsum = SUM of BORON densities
!  ctrlsum  = SUM of all ConTRoL variables
!  errsum   = SUM of mass residual ratio and ERRor estimate
!  fluxsum  = SUM of neutron FLUXes
!  qualsaum = SUM of QUALities of noncondensAbles
!  psum     = SUM of ~pressures
!  rhshesum = SUM of Heat Equations RHS
!  rhsnksum = SUM of Neutron Kinetics system RHS
!  rhsthsum = SUM of TH RHS (semi- or nearly-implicit)
!  tempsum  = SUM of TEMPeratures in all heat structure geometries
!  tripsum  = SUM of all TRIP â€œtimeofâ€  values
!  ufsum    = SUM of Fluid (liquid) internal energies
!  ugsum    = SUM of Gas internal energies
!  vfsum    = SUM of Fluid (liquid) velocities
!  vgsum    = SUM of Gas velocities
!  voidsum  = SUM of VOID fractions
!
!  3.0  Internal Subroutines
!
 contains

 subroutine verf_CaseInit (ilowlimit, iuplimit, ncount)
!******************************************************
 integer :: ilowlimit, iuplimit
 integer (sik) :: ncount
!
 backtime = 0
 caseDmpCnt = 0
 dumpStep = -1
 ljunflag = .false.
 lvolflag = .false.
 verfaction = 0
 verfDumpLeft = iuplimit
!lverify = ilowlimit <= ncount .and. ncount <= iuplimit .and. verfaction > 0
 lverify = .false.
 call verf_Init
```

```
!****************************
 end subroutine verf_CaseInit

 subroutine verf_Dump (icount, advtime)
!****************************
!  Dump summed information on the verification file
 real (sdk) :: advtime
 integer (sik) :: icount
!
   caseDmpCnt = caseDmpCnt + 1
   verfspace = verfspace + 1305
   write (verifl,2000) caseDmpCnt, icount, advtime
   2000 format (/"Dump",i6, 4x,"Advancement=",i8," time=",1pe12.4)
   write (verifl,2010) "P=    ",vrf%psum,vrf%psum
   write (verifl,2010) "Uf=   ",vrf%ufsum,vrf%ufsum
   write (verifl,2010) "Ug=   ",vrf%ugsum,vrf%ugsum
   write (verifl,2010) "VOIDg=",vrf%voidgsum,vrf%voidgsum
   write (verifl,2010) "QUALa=",vrf%qualasum,vrf%qualasum
   write (verifl,2010) "Boron=",vrf%boronsum,vrf%boronsum
   write (verifl,2010) "Vf=   ",vrf%vfsum,vrf%vfsum
   write (verifl,2010) "Vg=   ",vrf%vgsum,vrf%vgsum
   write (verifl,2010) "RHSth=",vrf%rhsthsum,vrf%rhsthsum
   write (verifl,2010) "SOLth=",vrf%thsolnsum,vrf%thsolnsum
   write (verifl,2010) "Error=",vrf%errsum,vrf%errsum
   write (verifl,2010) "Temp= ",vrf%tempsum,vrf%tempsum
   write (verifl,2010) "Flux= ",vrf%fluxsum,vrf%fluxsum
   write (verifl,2010) "dtsum=",vrf%dtsum,vrf%dtsum
   write (verifl,2010) "Trips=",vrf%tripsum,vrf%tripsum
   write (verifl,2010) "Cntrl=",vrf%cntrlsum,vrf%cntrlsum
   2010 format (a6,1pe24.16,x,Z32)
   write (verifl,2020) vrf%rdccrnt, vrf%rdcextr, vrf%rdcmerr, vrf%rdcprop, vrf%rdcqual
   2020 format ("Rdc:Crnt,Extrp,Mass,Prop,Qual=", 5i6)
   write (verifl,2030) vrf%rptair, vrf%rptdpr, vrf%rptflip, vrf%rptjpack, vrf%rptpack
   2030 format ("Rpt:Air,DelP,Flip,Jpack,Vpack=", 5i6)
!************************
 end subroutine verf_Dump

 subroutine verf_Header (ctitle, ncase, opnd, ptitle)
!********************
!  Write header information on the verification file
 integer (sik) :: ncase
 logical :: opnd
 character (*) :: ctitle, ptitle
!  Locals
 integer (sik) :: nncase
!
!  1.0  Initialize
 if (ncase < 0) then
   nncase = -ncase
 else
   nncase = ncase
 endif
!
!  2.0  Output
 if (.not.opnd) then
   write (verifl,"(a,3x,a)") trim(ptitle(1:24)), trim(compilehost)
   write (verifl,1000) trim(compileday), trim(compiletime)
   1000  format ("Time compiled: ",a,x,a)
   write (verifl,1001) ctitle(82:102)
   1001  format ("Date and Time of run: ",a,x,a)
   verfspace = 56
 endif
 write (verifl,1002) nncase, trim(ctitle(1:80))
```

```
 1002  format (/"Case ",i2,2x,a)
 verfspace = verfspace + 9 + len_trim(ctitle(1:80))
 return
!************************
 end subroutine verf_Header

 subroutine verf_Init
!********************
!  Initialize components of verf derived type, the sum scalars.
 vrf%vfsum = 0.0
 vrf%vgsum = 0.0
 vrf%qualasum = 0.0
 vrf%boronsum = 0.0
 vrf%ufsum = 0.0
 vrf%ugsum = 0.0
 vrf%voidgsum = 0.0
!
 vrf%tempsum = 0.0
 vrf%fluxsum = 0.0
!
 vrf%thsolnsum = 0.0
 vrf%rhsthsum = 0.0
 vrf%rhshesum = 0.0
 vrf%rhsnksum = 0.0
!
 vrf%dtsum = 0.0
 vrf%errsum = 0.0
!
 vrf%cntrlsum = 0.0
 vrf%tripsum = 0.0
 vrf%rdccrnt = 0
 vrf%rdcextr = 0
 vrf%rdcmerr = 0
 vrf%rdcprop = 0
 vrf%rdcqual = 0
 vrf%rptair = 0
 vrf%rptdpr = 0
 vrf%rptflip = 0
 vrf%rptjpack = 0
 vrf%rptpack = 0
 return
!************************
 end subroutine verf_Init

 subroutine verf_Last (cputime)
!*********************
!  Write final information of current input case on the verification file
 real (sdk) :: cputime
!
 verfspace = verfspace + 46
 write (verifl,'(/"CPU Time=",1pe24.16," size",i8)') cputime,verfspace
!
!  Enforce 1 MB size restriction
!  The verification file may not exceed this size (except for final time step)
 if (verfspace > verfsizlim) then
   verfaction = -1
   lverify = .false.
   close (unit = verifl)
 endif
 return
!*************************
 end subroutine verf_Last
```

```
 subroutine jefvcahs (jc0, jc1, jc2)
!**********************************
 integer (sik) :: jc0, jc1, jc2
 ljunflag(1) = ljunflag(1) .or. btest(jc0,25)
 ljunflag(2) = ljunflag(2) .or. btest(jc1,15)
 ljunflag(3) = ljunflag(3) .or. btest(jc1,2)
 ljunflag(4) = ljunflag(4) .or. ibits(jc0,17,2) > 0
 ljunflag(5) = ljunflag(5) .or. .not.btest(jc0,4) .or. btest(jc2,10)
 ljunflag(6) = ljunflag(6) .or. btest(jc0,8) .or. btest(jc1,29)
 ljunflag(7) = ljunflag(7) .or. btest(jc0,9)
 ljunflag(8) = ljunflag(8) .or. ibits(jc0,12,2) > 0
!************************
 end subroutine jefvcahs

 subroutine tlpvbfe (imap, vct)
!*****************************
 integer (sik) :: imap, vct
 if (.not.btest(vct,1)) then
   lvolflag(1) = lvolflag(1) .or. btest(vct,2)
   lvolflag(2) = lvolflag(2) .or. btest(imap,28)
   lvolflag(3) = lvolflag(3) .or. .not.btest(vct,7)
   lvolflag(4) = lvolflag(4) .or. .not.btest(imap,9)
   lvolflag(5) = lvolflag(5) .or. btest(vct,30) .or. btest(imap,27)
   lvolflag(6) = lvolflag(6) .or. .not.btest(imap,13)
   lvolflag(7) = lvolflag(7) .or. btest(vct,1)
 endif
!************************
 end subroutine tlpvbfe

end module verifymod
```

6.2 RDEBUG Subroutine

Since so much of RDEBUG changed, it is simplest to present the entire subroutine. It processes all "199" input cards. It has been augmented to include:

- "199 verify"
- "199 noverify"

For "noverify," there are no other words on the card and it simply closes the verification file and shuts off output further output to it on subsequent input cases. For "verify" there are up to four words on either type of card.

The format 199 card format is:

"199 verify W2 W3 W4"

W2 is Action, W3 is Start time, advancement number, and W4 is End time or advancement.

- W2 = Action has the values: dump, backair, backpck, backvel, or backall.
- W3 = Start = timestep number (integer) or cumulative time (floating point) on which to activate the Action.
- W4 = End = Timestep number (if W3 and W4 are integers) on which to terminate the action. If W3 is real, W4 is the number of advancements to stay active. Special value -1 means to deactivate on the final timestep.

To have the verification dump only on the final step, set the start time greater than the end time.

```
      subroutine rdebug
!
!  DESCRIPTION
!    Read the input for debugging a subroutine.
!    NOTE:  Only DTSTEP at this time.
!  Co-Authors:  George Mesina, Richard Wagner
!  Cognizant:   George Mesina
!  Created:     July 31, 2008
!  Updated:     11/18,12 (GM), 3/18/13
!
!  DECLARATIONS
      use cctlmod
      use ctrlmod
      use gnrlmod, only: ctitle, ptitle
      use inputmod
      use testmod
      use ufilfmod, only: filsch
      use ufilsmod
      use verifymod
!
      implicit none
!  LOCALS
      real*8 :: rscr(4)
      integer l3a(10)
      integer (8) :: iscr(4)
      integer i, n2, n3, n4, n5
      integer, save :: indvrf=15, ios
```

```
      character (8) cscr(4)
      equivalence (cscr(1),iscr(1), rscr(1))
      data l3a /199, 0, 1, 4, 0, 1, -1, -1, 0, 0/
!
!  DATA DICTIONARY
!
!  cscr     Character SCRatch. contains the input line form INP.
!  iscr     Integer SCRatch. contains the input line form INP.
!  i        Index for do loops
!  l3a      INP instruction array
!     (1) = RELAP5-3D input card number (identification number)
!     (2) = final (last) card number. Zero means only one card.
!           Positive means card numbers must be consecutive.
!           Negative means they don't have to be consecutive.
!     (3) = minimum number of expected (allowed) arguments on card
!     (4) = maximum number of expected (allowed) arguments on card
!     (5) = skip factor (for storage in output array)
!     (6) = on input, index of first word in output array.
!           on output, -1 if error on card
!                      n>0 (number of words placed on output array
!     (7+) = Code for describing the type of input
!            0        means integer
!            1        means floating point
!            -1       means character
!            |n| > 1  repeat count
!                     n < -1   repeat starts at beginning for each new
!                              card (See manual.)
!                     n > 1    repeat continues from previous card.
!
!  Executable Code
!
!  Initialize
      ctest = 'null'
      l3a(6) = 1
      call inplnkn (l3a(1), n2, n3, n4)
      if (n4 == 0) return
!
!  Obtain 199-card input from INP storage
!  Determine if it is invalid or valid - 4 words of proper typing
      call inpmod (l3a,n3,n4,n5,0)
      if (n5 == 0) then
        call inp2n (iscr, l3a)
      else
!     if (l3a(6) < 0) then
        l3a(6) = 1
        l3a(9) = 1
        call inp2n (iscr, l3a)
      endif
      if (l3a(6) < 0) then
        fail = .true.
      else if (l3a(6) == 0) then
        return
      else if (trim(cscr(1)) == "noverify" .and. l3a(6) /= 1) then
        fail = .true.
        write (output,'("0******** Two input items are required on",
     &   " a 199 no-verify card.")')
      else if (trim(cscr(1)) == "verify" .and. l3a(6) < 3 .or.
     & l3a(6) > 4) then
        fail = .true.
        write (output,'("0******** Four input items are required on",
     &   " a 199 card.")')
      else
!
```

```
!  Valid 199 card format
!  Word 1 in {verify, noverify, debug}
!  Word 2 in {backall, backair, backpck, backvel, dump}
!  Word 3 = (REAL) activation cumulative time
!           (INT, ACT<5) attempted advancement on which to activate
!           (INT, ACT<5) successful advancement on which to activate
!  Word 4 = (INT)  > 0      shutdown Advancement
!           (INT)  -1       shutdown at end of transient
!  Summary
!  (W3, W4) = (R, I)  start cumulative time, number of steps to take
!           = (R, -1) start cumulative time, shutdown at transient end
!           = (I, I)  start and end Advancement (see action value)
!           = (I, -1) start Advancement, shutdown at transient end
!
!  Translation into variables
!  beginVerf = BEGINning VERiFy cumulative time on which to activate
!              -1.0  if  integer-trigger used & active-to-transient-end
!  ilowlimit = Integer LOWer LIMIT to activate verification
!              set to zero if Word 3 is REAL.
!  iuplimit  = Integer UPper LIMIT to shutdown verification
!              -1  if  active-to-transient-end
         ctest = cscr(2)
         iuplimit = iscr(4)
         if (iuplimit > 0) then
!  W4 > 0
           if (l3a(9) == 1) then
!  W3 real
             beginVerf = rscr(3)
             ilowlimit = iuplimit + 1
           else
!  W3 int
             beginVerf = -1.0
             ilowlimit = iscr(3)
           endif
         else
!  W4 < 0
           ilowlimit = 0
           beginVerf = -1.0
         endif
         call to_lower(cscr(1))
         select case (trim(cscr(1)))
         case ("dtstep")
           call proc_debug
         case ("verify", "noverify")
           call proc_verify
         case default
           fail = .true.
           write (output,'("0******** Invalid 199-card keyword.") ')
         end select
       endif   !l3a
       return
      contains
       subroutine to_lower (string)
       character*(*) string
       integer :: i, icode, lacode, uacode, uzcode
       lacode = ICHAR('a')
       uacode = ICHAR('A')
       uzcode = ICHAR('Z')
       do i = 1, LEN_TRIM(string)
         icode = IACHAR(string(i:i))
         if (icode >= uacode .and. icode <= uzcode) then
           string(i:i) = CHAR(icode + lacode - uacode)
         endif
```

```
      end do
      return
      end subroutine to_lower
      subroutine proc_debug
!     ********************
      read (ctest(3:4),"(a2)") Aset
      read (ctest(1:2),"(i2)",iostat=ios) testNo
      if (ios /= 0) then
        write (output,'("0******** rdebug: Invalid Test Number ",
     &     a2)') ctest(1:2)
        fail = .true.
      else
        if (testNo<0 .or. testNo>17 .or. Aset/="A1" .and. Aset/="A2")
     &     then
          write (output,'("0******** Quitting. Bad test code: ",a4)')
     &     ctest
          fail = .true.
        endif   !testNo
        if (ilowlimit<1 .or. ilowlimit>iuplimit) then
          write (output,'("0******** Invalid Advancement Limits ",
     &     2i12)') ilowlimit, iuplimit
          fail = .true.
        endif   !ilowlimit
!
        if (testNo < 10) then
          ctest(1:2) = cbasic(testNo)
          ctest(3:6) = " "
        else
          ctest(1:2) = "a1"
          if (testNo < 14) then
            ctest(3:4) = cbasic(testNo-8)
            ctest(5:6) = "  "
          else
            ctest(3:4) = cbasic(testNo-12)
            ctest(5:6) = "d1"
          endif
        endif   !testNo
      endif   !ios
      return
!     ************************
      end subroutine proc_debug
      subroutine proc_verify
!     **********************
!  Process the verification file input cards, set data accordingly, and
!  open or close the verification file. Legitimate options:
!  199 verify dump start stop   (start=real or int, stop=int)
!  199 verify backair
!  199 verify backpck
!  199 verify backvel
!  199 noverify
!
!
!  DECLARATIONS -  Local
      implicit none
      integer :: openerr
      integer (sik) :: nvdump
      logical :: opnd, there
!
!  Executable Code
!
!  1.0  Initialize
      nvdump = 0
      opnd = .false.
```

```
       there = .false.
!**********************************************************************
       call verf_CaseInit (ilowlimit, iuplimit, nvdump)
!**********************************************************************
       inquire (file = filsch(indvrf), opened = opnd, exist = there)
!
!  2.0  Process Verify or Noverify
!
!  2.1  "199 Noverify"
!  Deactivate verification and close verify file
       if (trim(cscr(1)) == "noverify") then
         inquire (file = trim(filsch(indvrf)), opened = opnd)
         verfaction = -1
         if (opnd) close (unit = verifl)
       else
!
!  2.2  "199 verify"
!
!  2.2.1  Interpret VERIFY "action" keyword
         select case (trim(ctest))
         case ("dump")
           verfaction = 1
         case ("backair")
           verfaction = 2
         case ("backpck")
           verfaction = 3
         case ("backvel")
           verfaction = 4
         case ("backall")
           verfaction = 5
         case default
           verfaction = 0
           fail = .true.
           write (output,2000) ctest
 2000    format ("0******** Invalid 199 verify card action value: ",a8)
         end select
       endif
!
!  2.2.2  Open Verification file
       if (verfaction > 0) then
         if (opnd) then
!  Already open - just write header
           call verf_Header (ctitle, ncase, opnd, ptitle)
         else if (.not. (opnd .or. there) ) then
!  Not open, non-existent - open
           open (unit = verifl, file = filsch(indvrf),iostat=openerr)
           if (openerr > 0) then
             write (output,*) "0******** Cannot open verify file:",
     &       trim(filsch(indvrf))
             fail = .true.
           else
             call verf_Header (ctitle, ncase, opnd, ptitle)
           endif
         else if (there .and. .not.opnd) then
!  Not open, existent.
!  According to the "zen of RELAP" this should be an error. However,
!  code users requested that the file be overwritten rather than have
!  RELAP5 fail with an error message.
           write (output,2010) trim(filsch(indvrf))
           write (*,2010) trim(filsch(indvrf))
 2010      format ("0$$$$$$$$ WARNING: Existing verifification file ",
     &     "overwrite requested for file: ",a)
           open (unit = verifl, file = filsch(indvrf))
```

```
            call verf_Header (ctitle, ncase, opnd, ptitle)
          endif
        endif
        return
!       *************************
        end subroutine proc_verify
        end subroutine rdebug
```

6.3 Subroutine VERFSUM

This subroutine performs all the L_1 norms that are recorded on the verification file in accordance with Equation (2.4.1). One internal subroutine for each group of data: Thermal Hydraulic, Temperature, Neutron Kinetics, Trips, Control Variables, and Code Statistics. There is a logical function subprogram that activates the subroutines, through a logical variable, only on the final step of the transient and on user requested steps.

```
      subroutine verfsum (typesum)
!
!  Description: Sum important data for the verification file and write
!               the data at the end of the final time step and user
!               requested steps.
!  Cognizant:   Dr. George L Mesina
!  Created:     Nov 13, 2012
!  Updated:     Nov 18, 2012
!
!  Declarations
!  Global
      use asdmod,  only: s_stscpu
      use ctrlmod, only: done, dt, fail, imdctl, iroute, ncount,
     &                   timehy
      use junmod,  only: jct, njct
      use ufilsmod, only: output, verifl
      use verifymod
      use volmod,  only: vlm, nvlm
!  Local
      implicit none
      real (sdk), save :: totaltime = 0.0
      character(*) typesum
!
!  Executable Code
!
!  1.0  Summations
!
!  1.1  Perform requested summation
      select case (typesum)
      case ('convar')
        call verfsum_cnv
      case ('hydro')
        call verfsum_th
      case ('kinetic')
        call verfsum_kin
      case ('lverify')
        lverify = verf_time (timehy, ncount)
        if (lverify) call verf_Init
      case ('restep')
        call verfsum_restep
      case ('rhsth')
        call verfsum_rhsth
      case ('temps')
        call verfsum_temps
      case ('thsoln')
        call verfsum_thsoln
      case ('trip')
        call verfsum_trips
      case default
        fail = .true.
```

```
        done = -100
      end select
!
!  1.2  Cumulative time sum
      if (done /= 0 .or. fail) then
        totaltime = totaltime + s_stscpu
      endif
!
!  2.0  Dump to Verification File
!  When all data is collected, the keyword in TYPESUM is RESTEP, dump
!  data for this advancement if:
!  * did not previously dump on this advancement (dumpStep contols this)
!    OR
!  * it is the final advancement (DONE is non-zero or FAIL).
      if (typesum == 'restep' .and. (dumpStep/=ncount)) then
        if (verfaction > 0 .and. (fail .or. done/=0)) then
          call verfsum_cnv
          call verfsum_th
          call verfsum_thsoln
          call verfsum_rhsth
          call verfsum_kin
          call verfsum_restep
          call verfsum_temps
          call verfsum_trips
          call verf_Dump (ncount, timehy)
          call verf_Last (totaltime)
          dumpStep = ncount
        else
          if (lverify .and. verfspace < verfsizlim) then
            if (4 >= verfaction .and. verfaction >= 1
     &       .and. timehy >= beginVerf) then
              call verf_Dump (ncount, timehy)
              dumpStep = ncount
              verfDumpLeft = verfDumpLeft - 1
            endif
          endif
        endif
      endif
!
      return

     contains

      subroutine verfsum_cnv
!     **********************
!  Sum of Control Variables
      use cnvmod, only: ncnvr, cnvr
      integer :: i
!
      vrf%cntrlsum = 0.0
      do i = 1, ncnvr
        vrf%cntrlsum = vrf%cntrlsum + abs(cnvr(i)%arn)
      enddo
!     *************************
      end subroutine verfsum_cnv

      subroutine verfsum_kin
!     **********************
!  Sum of neutron fluxes
      use k3allmod, only: ismeth, ngrk, nxytmax, nz
      use k3dmod, only: sw
      use kinmod, only: rk_opt, rk_pow
#ifdef use_phisics
```

```
      use instmod, only: need_instant
#else
      logical :: need_instant = .false.
#endif
      integer i, j, k
!
      vrf%fluxSum = 0.0
      if (.not.btest(rk_opt,7)) then
!  Point
        vrf%fluxSum = rk_pow
      else if (ismeth == 1 .or. need_instant) then
!  Krylov
        do i = 1, nxytmax
          do j = 1, nz
            do k = 1, ngrk
              vrf%fluxSum = vrf%fluxSum + abs(sw(i,j,k))
            end do
          end do
        end do
      endif
      return
!     *************************
      end subroutine verfsum_kin

      subroutine verfsum_restep
!     *************************
!  Count time step reattempts caused by repeat and reduction conditions,
!  error measures, and time data
      use asdmod, only: as1, as2, s_stsdtn, s_stsdtx
      use ctrlmod
!
      implicit none
      integer airsum, dprsum, packsum
      integer flipsum, jpacksum
      integer crntsum, extrsum, merrsum, propsum, qualsum
      integer mk
!
!  1.0  Repeats w/o dt halving
!  1.1  Volume-based
!  Sum up air appearance, pressure change, and water packing repeat
      airsum = 0
      dprsum = 0
      packsum = 0
      do mk = 1, nvlm
        airsum = airsum + abs(as1(mk)%strap1)
        dprsum = dprsum + abs(as1(mk)%strdp1)
        packsum = packsum + abs(as1(mk)%stvpk1)
      end do   !mk
      vrf%rptair = airsum
      vrf%rptpack = packsum
      vrf%rptdpr = dprsum
!
!  1.2  Junction-based
!  Sum up water packing and velocity flip/flop repeats
      jpacksum = 0
      flipsum = 0
      do mk = 1, njct
        jpacksum = jpacksum + abs(as2(mk)%stjpk1)
        flipsum = flipsum + abs(as2(mk)%stjff1)
      end do   !mk
      vrf%rptjpack = jpacksum
      vrf%rptflip = flipsum
!
```

```
!  2.0  Reduction and reattempt
!  Sum up quality, extrapolation, mass error, fluid property, and
!  material Courant violations that cause reduction and repeat
!  *** For restart, only use value since lage major edit, such as
!      strcl1, but not cumulative, such as strcl2.
         qualsum = 0
         extrsum = 0
         merrsum = 0
         propsum = 0
         crntsum = 0
         do mk = 1, nvlm
           crntsum = crntsum + abs(as1(mk)%strcl1)
           extrsum = extrsum + abs(as1(mk)%strex1)
           merrsum = merrsum + abs(as1(mk)%strte1)
           propsum = propsum + abs(as1(mk)%strpe1)
           qualsum = qualsum + abs(as1(mk)%strxl1)
         end do   !mk
         vrf%rdccrnt = crntsum
         vrf%rdcextr = extrsum
         vrf%rdcmerr = merrsum
         vrf%rdcprop = propsum
         vrf%rdcqual = qualsum
!
!  3.0  Time  and error data
!      vrf%dtsum = s_stsdtn + s_stsdtx + dt + dthy + dtkn
       vrf%dtsum = dt + dthy + dtkn
       vrf%errsum = abs(errmax) + abs(emass/max(tmass,tmasso,1.0e-20))
       return
!      ************************
       end subroutine verfsum_restep

       subroutine verfsum_th
!      *********************
!  Sum the primitive variables of the governing equations across all
!  hydrodynamic systems for the verification file:
!  Pressure, phasic internal energies, noncondensable quality, gas void
!  fraction, boron density, and phasic velocities.
!  NOTE: The scalar summands are QUADRUPLE PRECESION to avoid roundoff.
       integer :: i
!
       vrf%psum = 0.0
       vrf%ufsum = 0.0
       vrf%ugsum = 0.0
       vrf%qualasum = 0.0
       vrf%voidgsum = 0.0
       vrf%boronsum = 0.0
       do i = 1, nvlm
         vrf%psum = vrf%psum + abs(vlm(i)%p)
         vrf%ufsum = vrf%ufsum + abs(vlm(i)%uf)
         vrf%ugsum = vrf%ugsum + abs(vlm(i)%ug)
         vrf%qualasum = vrf%qualasum + abs(vlm(i)%quala)
         vrf%voidgsum = vrf%voidgsum + abs(vlm(i)%voidg)
         vrf%boronsum = vrf%boronsum + abs(vlm(i)%boron)
       end do
!
       vrf%vfsum = 0.0
       vrf%vgsum = 0.0
       do i = 1, njct
         vrf%vfsum = vrf%vfsum + abs(jct(i)%velfj)
         vrf%vgsum = vrf%vgsum + abs(jct(i)%velgj)
       end do
       return
!      ************************
```

```
      end subroutine verfsum_th

      subroutine verfsum_thsoln
!     ************************
!  This only calculates the norm for (bpi,bpj) = (1,1).
!  *** This could be expanded to other TH systems in the future.
!  The sourcem array is overwritten by subsequent systems. Save
!  sourcem array in verfsol array for recalculation at the end of
!  transient.
      use bpmod
      real*16 :: solnsum
      integer :: i, nvar
      logical :: doNorm
!
      nvar = msiz(1)%nequ(1)
      if (typesum=='thsoln'.and.bpi==1.and.bpj==1) then
        verfsoln(1:nvar) = sourcem(1:nvar)
      endif
      if (done /= 0 .or. fail) then
        solnsum = 0.0
        sourcem(1:nvar) = verfsoln(1:nvar)
        do i = 1, nvar
          solnsum = solnsum + abs(sourcem(i))
        end do
        vrf%thsolnsum = solnsum
      else if (typesum=='thsoln'.and.bpi==1.and.bpj==1) then
        solnsum = 0.0
        do i = 1, nvar
          solnsum = solnsum + abs(sourcem(i))
        end do
        vrf%thsolnsum = solnsum
      endif
!GLM!  call manualvrf('thsoln')
      return
!     ****************************
      end subroutine verfsum_thsoln

      subroutine verfsum_rhsth
!     ************************
!  This only calculates the norm for (bpi,bpj) = (1,1).
!  *** This could be expanded to other TH systems in the future.
!  Save the RHS in verfrhs array for recalculation at the end of
!  transient.
      use bpmod
      real*16 :: rhsth
      integer :: i, nvar
      logical :: doNorm
!
      nvar = msiz(1)%nequ(1)
      if (typesum=='rhsth'.and.bpi==1.and.bpj==1) then
        verfrhs(1:nvar) = sourcem(1:nvar)
      endif
      if (done /= 0 .or. fail) then
        rhsth = 0.0
        sourcem(1:nvar) = verfrhs(1:nvar)
        do i = 1, nvar
          rhsth = rhsth + abs(sourcem(i))
        end do
        vrf%rhsthsum = rhsth
      else if (typesum=='rhsth'.and.bpi==1.and.bpj==1) then
        rhsth = 0.0
        do i = 1, nvar
          rhsth = rhsth + abs(sourcem(i))
```

```
          end do
          vrf%rhsthsum = rhsth
        endif
!GLM!   call manualvrf('rhsth')
!       ************************
        end subroutine verfsum_rhsth

        subroutine verfsum_temps
!       ************************
!   Sum heat structure geometry temperatures array.
        use hsgmod, only: nhtsgs, htg
        integer :: cols, its, mm, nh
!   Array %httmp has subscripts; (Rows, Cols, Old/New)
!   *  where 0=Old, 1=New
        vrf%tempsum = 0.0
        do nh = 1, nhtsgs
          cols = htg(nh)%geo%htnmpt
          do its = 1, htg(nh)%geo%htnusr
            do mm = 1, cols
              vrf%tempsum = vrf%tempsum + abs(htg(nh)%httmp(its,mm,1))
            end do !mm
          end do !its
        end do !nh
        return
!       ***************************
        end subroutine verfsum_temps

        logical function verf_time (advtime, icount)
!       *********************************
!   Determine if verification sums should be calculated this advancement
!   CONSIDERATION
!   This depends on the type of verification.
!   * On forced backups, only the successful advancements are considered.
!   * Otherwise, "ncount" total advancements are used.
        use asdmod, only: s_strdc1, s_strdc2
        use idtmod, only: dtsmallest
        use testmod, only: ilowlimit, iuplimit
        use ufilsmod
        integer icount
        real (sdk) :: advtime
!   Local
        integer (sik) :: jadv
        real (sdk), parameter :: eps = 1.0e-12
!   jadv = number of successful advancements (if verfaction is 5)
!        = total advancements                (if 0 < verfaction < 5)
!
        verf_time = .false.
        if (verfspace < verfsizlim .and. verfaction > 0) then
          if (beginVerf < 0.0) then
            if (verfaction == 5) then
              jadv = icount - (s_strdc1 + s_strdc2)
            else
              jadv = icount
            endif
            verf_time = ilowlimit <= jadv .and. (jadv <= iuplimit .or.
     &      iuplimit == -1)
          else if (advtime >= beginVerf - dtsmallest/4 .and.
     &    verfDumpLeft > 0) then
            verf_time = .true.
! ADD initialization of iupcount = iuplimit (+1)
          endif
        endif
!       **********************
```

```
      end function verf_time

      subroutine verfsum_trips
!     ************************
!  Sum the trips timeof values (other choices would mix units).
      use trpmod
      real (sdk) :: xl1, xl2
      integer :: i
!
      vrf%tripsum = 0.0
!
!  1.0  Variable Trip TIMEOF values
      do i = 1, ntrpv
        if (iroute/=1 .or. iroute==1.and.btest(imdctl(1),6)) then
          vrf%tripsum = vrf%tripsum + abs(trpv(i)%trptim)
        else
          vrf%tripsum = vrf%tripsum + abs(trpv(i)%trptimss)
        endif
      end do
!
!  2.0  Logical Trip TIMEOF values
      do i = 1, ntrpl
        if (iroute/=1 .or. iroute==1.and.btest(imdctl(1),6)) then
          vrf%tripsum = vrf%tripsum + abs(trpl(i)%trptim)
        else
          vrf%tripsum = vrf%tripsum + abs(trpl(i)%trptimss)
        endif
      end do
      return
!     ***************************
      end subroutine verfsum_trips

      end subroutine verfsum
```

6.4 Subroutine VERFBACKUP

Short subroutine VERFBACKUP controls the backup on every advancement logic.

```
subroutine VerfBackup
!
!  Set backup flags at the hydrodynamic system (or loop) level to force
!  backups at a given time or advancement based on verfaction.
!  Do not force a fake backup if succes is already 5.
!  Do not force a fake backup more than once at a given time.
!
 use intrtype
 use ctrlmod, only: ncount, succes, timehy
 use idtmod
 use lpdmod, only: lpd
 use testmod, only: ilowlimit, iuplimit
 use ufilsmod, only: output
 use verifymod, only: backtime, beginVerf, verfaction, verfDumpLeft
!
 implicit none
 logical   :: lforce, ltimeAdv
!
!  Executable Code
!
 ltimeAdv = (timehy > backtime + dtsmallest/4)
 backtime = timehy
 if (ltimeAdv .and. succes /= 5) then
   lforce = ltimeAdv .and. verfaction==5
   if (lforce .or. timehy >= beginVerf .and. verfDumpLeft > 0 .or.     &
       ilowlimit <= ncount .and. ncount <= iuplimit) then
     select case (verfaction)
     case (2)
       lpd(1)%airap = 1
       write (output,'("Forcing air appearance backup on step",i9)') ncount
     case (3)
       lpd(1)%lpackr = 1
       lpd(1)%wpack = 1
       write (output,'("Forcing water packing backup on step",i9)') ncount
     case (4)
       lpd(1)%vlflip = 1
       write (output,'("Forcing velocity flip-flop backup on step",i9)') ncount
     case (5)
       lpd(1)%airap = 1
       write (output,'("Forcing backup (2 forward / 1 back) on step",i9)') ncount
     end select
!
!    succes = 5
     lpd(1)%lsuces = 5
   endif
 endif
!
 return
 end subroutine VerfBackup
```

6.5 Minor Modifications of Existing Subroutines

Modifications to existing routines were kept small to reduce the intrusion of new code which is and mostly tangential to the main purpose of those subroutines. In the past programmers have made extensive changes to existing routines, often surrounded by pre-compiler protection, this interferes with both the readability and understandability of the subroutine's operations. Therefore, modifications were kept to a minimum.

Temporarily for the convenience of the recipients the changes are marked within the source code with the comment:

```
!!!! verification !!!!
```

The comment occurs both above and below the verification coding. This marking via comments will not persist in the main line of RELAP5-3D. It should be removed after the coding is introduced.

6.5.1 Subroutine DTSTEP Modifications

Modifications to subroutine DTSTEP are complicated by the need to perform differently in restart and non-restart.

A User Problem was discovered and recorded when the ANS problem was restarted at a timecard endpoint rather than at a point in the middle of a time interval, while all the timecards except those ending before the restart time were omitted. Due to a combination of programming issues, the timestep from the timecard ending at the restart point was used on restart. The code took 10x as many steps as needed. It nonetheless arrived at the final time with EXACTLY the same verification dump, except for number of timesteps and timestep sum.

To correct this, DTSTEP handles initialization for restart differently from a new problem, both in its sections 1 and 4. It also makes a call to VERFSUM just before exiting in its section 11. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod
```

Section 1.3, 3rd line

```
        if (verfaction > 0) call verfsum ('lverify')
```

Immediately above 11.4.2

```
!  11.4.1.2  Verification Dump
        if (verfaction > 0 .and. succes /= 5) then
          call verfsum ('restep')
        endif
!GLM!   call manualvrf ('total')
```

6.5.2 Subroutine HYDRO Modifications

Modifications to subroutine HYDRO are made to sum primitive variables and to implement backup testing. A conditional call to VERFSUM is made just above the call to HTFINL. A conditional call to VERFBACKUP is made immediately after each of the velocity routines VFINL and VIMPLT. Note that these calls are not made on the first attempted timestep due to certain algorithmic difficulties. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod , only: lverify, verfaction, vrf
```

Section 1.3, 3rd line

```
!  Verification file:
      if (lverify) then
        if (verfaction > 0 .and. succes /= 5) then
!  Sum primitive vars of governing equs on non-backup
          call verfsum ('hydro')
```

```
        endif
      endif
```

6.5.3 Subroutine IEDIT Modifications

Modifications to subroutine IEDIT are limited to creating information on the output file to aid the building of the verification "features-decks" matrix. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod
```

After enddo below format 2008

```
!  Verification File calculations and output
      if (lverify) then
        do i = 1, nvlm
          call tlpvbfe (vlm(i)%imap(1), vlm(i)%vctrl)
        enddo
        write (output,*) "Volume flags on in any volume"
        if (lvolflag(1)) write (output,*) "vol-flag on: thermal strat"
        if (lvolflag(2)) write (output,*) "vol-flag on: mixture level"
        if (lvolflag(3)) write (output,*) "vol-flag on: water packing"
        if (lvolflag(4)) write (output,*) "vol-flag on: vertical strat"
        if (lvolflag(5)) write (output,*) "vol-flag on: bundle"
        if (lvolflag(6)) write (output,*) "vol-flag on: wall friction"
        if (lvolflag(7)) write (output,*) "vol-flag on: equilibrium"
      endif
!  End of Verification Coding
```

Above return statement

```
!  Verification Coding
      if (lverify) then
        write (output,*) "Junction flags on in any junction"
        do j = 1,njct
          call jefvcahs (jct(j)%jc, jct(j)%jcex, jct(j)%jcex2)
        end do
        if (ljunflag(1)) write (output,*) "jun-flag on: jet junction"
        if (ljunflag(2)) write (output,*) "jun-flag on: modified PV"
        if (ljunflag(3)) write (output,*) "jun-flag on: CCFL"
        if (ljunflag(4)) write (output,*) "jun-flag on: HSE"
        if (ljunflag(5)) write (output,*) "jun-flag on: choking"
        if (ljunflag(6)) write (output,*) "jun-flag on: abrupt area"
        if (ljunflag(7)) write (output,*) "jun-flag on: homogeneous"
        if (ljunflag(8)) write (output,*) "jun-flag on: momentum flux"
      endif
!  End of Verification Coding
```

6.5.4 Subroutine INITDATA Modifications

Modifications to subroutine INITDATA initialize variables for verification processes. There is a call to a verification module subroutine and setting verifl to 18 and adding verify to array filsch properly. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod
```

Below call k3tctlinit

```
     call verf_CaseInit (0, 0, 0)
```

In "c Data for ufiles common return statement"

```
        data input/11/,output/12/,rstplt/13/,stripf/14/,plotfl/15/,
     & sth2xt/16/,jbinfo/17/,eoin/51/,verifl/18/,coupfl/19/,inpout/20/,
     & nodpower/21/,force/23/,fluxdata/24
```

In "c Data for ufilef common."

```
        data filsch/'ftb1','indta','outdta','plotfl','restrt','stripf',
     & 'jbinfo','null','cdffile','coupfl',' ','0','dumpfil1','dumpfil2',
     & 'verify',' ','r5-r5f',' '
```

6.5.5 Subroutine RTSC Modifications

Modifications to subroutine SYSSOL are limited to using the new module, initializing the variables that hold the sum, and calling the summation routine, VERFSUM. Usage of module VERIFYMOD was added to the declarations section. Initialization occurs at the very top in the initialization section of SYSSOL. The call to VERFSUM was placed after the call to the solver subroutines. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod, only: lverify
```

Among the "Local variables" add k2res to end of integer statement.

```
      integer i, ix, j, k0, k1, k2, k2res
```

Above "tsc(j)%tsppac = ior(ior(ishft(k1, 12), ishft(k0, 6)), k2)"

```
!  Verification File Info
      if (lverify) then
        k2res = mod(k2, 16)
        if (k2res == 3) write (output,*) "Advancement tt=3"
        if (k2res == 7) write (output,*) "Advancement tt=7"
        if (k2res == 11) write (output,*) "Advancement tt=11"
        if (k2res == 15) write (output,*) "Advancement tt=15"
      endif
!  End of Verification File Info
```

6.5.6 Subroutine SYSSOL Modifications

Modifications to subroutine SYSSOL are limited to using the new module, initializing the variables that hold the sum, and calling the summation routine, VERFSUM. Usage of module VERIFYMOD was added to the declarations section. Initialization occurs at the very top in the initialization section of SYSSOL. The call to VERFSUM was placed after the call to the solver subroutines. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod, only: lverify , verfaction, vrf
```

Above MA18 comment.

```
      if (lverify) then
        call verfsum('rhsth')
      endif
```

Above "Solver measures"

```
!  Verification
      if (lverify) then
        call verfsum ('thsoln')
      endif
```

6.5.7 Subroutine TRAN Modifications

Modifications to subroutine TRAN are limited to using the new module and calling the summation routines in the appropriate places. Usage of module VERIFYMOD was added to the declarations and calls to VERFSUM were placed after calls to subroutines that produced the arrays to sum for the verify dumps. TRAN calls VERFSUM at the top of the time loop to set lverify (which is only true when the array sums should be calculated). It also calls VERFSUM after the calls to trip, hydro, rkin, and convar. The changes are located as follows:

Declarations (in alphabetical order)

```
      use verifymod, only: lverify , verfaction, vrf
```

Below 200 continue.

```
      if (verfaction > 0) call verfsum ('lverify')
```

Below "call trip"

```
        if (lverify) call verfsum ('trip')
```

Below "call hydro"

```
      if (lverify) call verfsum ('temps')
```

Above "End of advanced reactor kinetics section"

```
        if (lverify) call verfsum ('kinetic')
```

Below "call convar"

```
        if (lverify) call verfsum ('convar')
```

6.5.8 Subroutines UFILFMOD and UFILSMOD

In UFLIFMOD, the default name of the verify file is loaded into the fname array at position 15.

In UFLISMOD, the variable verifl is created, documented and given the value 18.

Declarations (in alphabetical order)

```
      integer (sik) :: coupfl,eoin,inpout,input,jbinfo,output,plotfl,
     & restrt,rstplt,statfl,sth2xt,stripf,tty,nodpower,force,
     & fluxdata,ikdtunit,verifl
```

Comments

```
!  verifl   File number for the VERIfication FiLe
```

Below "jbinfo = 17"

```
      verifl = 18
```